A Rate-Based Congestion Control Framework for Connectionless Packet-Switched Networks

A thesis submitted to the
School of Computer Science
University College
University of New South Wales
Australian Defence Force Academy
for the degree of Doctor of Philosophy

By
Warren Keith Toomey
December 1997

Certificate of Originality

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person, nor material which to a substantial extent has been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgment is made in the thesis. Any contribution made to the research by colleagues, with whom I have worked at UNSW or elsewhere, during my candidature, is fully acknowledged.

I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.

Warren Keith Toomey

Abstract

The control of congestion in connectionless packet-switched wide-area networks is a major problem, especially in such networks as the Internet, which is experiencing an exponential growth in users and network traffic. This thesis outlines a rate-based framework for congestion control in these networks, examines the requirements of the framework, and describes a number of control mechanisms which meet the framework's requirements.

The effectiveness of the framework in combatting congestion is demonstrated by a series of simulation experiments, which also compare the framework against traditional end-to-end congestion control mechanisms. Experimental results indicate that rate-based congestion control provides excellent congestion control. Moreover, a rate-based framework achieves much better congestion control in these networks than traditional control mechanisms.

Acknowledgments

I would like to express my gratitude to my supervisor, George Gerrity, and my co-supervisor, Lawrie Brown, for their criticisms, insights, thoughtful suggestions and continual prodding during the course of this study.

This thesis is dedicated to my fiancée Lynne Smith: without her support and encouragement, this work may not have been completed.

This thesis is set in Adobe Palatino, Computer Modern Typewriter and Computer Modern Math using LaTeX 2$_\varepsilon$ and dvips version 5.58. Figures were drawn with xfig version 3.1.3, and converted to PostScript. Graphs were created with Gnuplot version 3.5 and converted through xfig to PostScript.

Contents

Certificate of Originality i

Abstract ii

Acknowledgments iii

1 Introduction 1

1.1 Connectionless Packet-switched Networks 1
1.2 Router Operation 2
1.3 Congestion in Connectionless Packet-switched Networks 4
1.4 Congestion Control Schemes 5
1.5 Factors Affecting Network Congestion 5
1.6 Scope of This Thesis 5

2 Existing Congestion Control Schemes 6

2.1 Jain's Congestion Survey 6
2.2 Traditional Control Schemes 8
2.3 Transmission Control Protocol 9
2.3.1 Sliding Window Flow Control 9
2.3.2 Sliding Window Congestion Control 9
2.3.3 Retransmission Timer Deficiencies 11
2.3.4 TCP Vegas 11
2.3.5 Summary 12
2.4 Network-based Congestion Control 13
2.4.1 ICMP Source Quench 13
2.4.2 The Dec Bit Mechanism 14
2.4.3 The Q-Bit Mechanism 16
2.4.4 Summary 16

2.5 Packet Discarding Mechanisms . . . 16
2.5.1 Random Drop . . . 17
2.5.2 Random Early Detection . . . 18
2.5.3 Summary . . . 18
2.6 Packet Queueing Mechanisms . . . 18
2.6.1 Nagle's Fair Switching . . . 18
2.6.2 Fair Queueing . . . 19
2.6.3 Summary . . . 19
2.7 Summary . . . 19

3 Rate-Based Congestion Control Schemes 22
3.1 Introduction . . . 22
3.2 Rate-Based Flow Control . . . 22
3.2.1 NETBLT . . . 22
3.2.2 XTP . . . 23
3.2.3 VMTP . . . 23
3.2.4 Summary . . . 24
3.3 Available Bit Rate Congestion Control in ATM Networks . . . 24
3.3.1 BECN . . . 24
3.3.2 PRCA . . . 24
3.3.3 EPRCA . . . 25
3.3.4 Summary . . . 25
3.4 Charney's Thesis . . . 26
3.5 Theoretical Work . . . 27
3.5.1 Summary . . . 28
3.6 Non-Service Degradation Control Schemes . . . 28
3.7 Summary . . . 29

4 A Rate-Based Framework for Congestion Control 31
4.1 Deficiencies and Remedies . . . 31
4.1.1 End-to-End Control Mechanisms . . . 31
4.1.2 Congestion Detection Schemes . . . 32
4.1.3 Router Procedures . . . 32
4.1.4 Packet Admission Mechanisms . . . 33
4.2 A Rate-Based Framework: Design Considerations . . . 33

4.3 Overview of the Framework . . . 34
4.4 Network Layer Rate Measurement . . . 35
4.5 Source Flow Control and Packet Admission Functions . . . 36
4.6 Source Error Control Function . . . 37
4.7 Destination Acknowledgment Function . . . 37
4.8 Router Sustainable Rate Measurement Function . . . 38
4.9 Router Packet Queueing Function . . . 38
4.10 Router Packet Dropping Function . . . 38
4.11 Router Congestion Avoidance . . . 38
4.12 Router Congestion Recovery . . . 38
4.13 Summary . . . 39

5 TRUMP **40**
5.1 Features of TRUMP . . . 41
5.1.1 Implicit and Explicit Connection Setup and Teardown . . . 41
5.1.2 Connection Identifiers . . . 41
5.1.3 Selective Retransmission . . . 42
5.1.4 Qualities of Service . . . 42
5.1.5 Dynamic Fragmentation and MTU . . . 42
5.1.6 Flow Control and Throughput . . . 42
5.1.7 Unacknowledged Data . . . 43
5.1.8 Integration with Traditional and New Communication Systems . . . 43
5.1.9 Security . . . 43
5.2 Connection Setup . . . 43
5.3 Data Transmission . . . 45
5.3.1 Destination Operation . . . 46
5.3.2 Source Operation . . . 47
5.4 Connection Teardown . . . 48
5.5 Abnormal Connection Termination . . . 49
5.5.1 End-to-End Flow Control . . . 49
5.6 Summary . . . 50

6 RBCC – Measuring a Source's Sustainable Rate **51**
6.1 Requirements of The Sustainable Rate Measurement Function . . . 51
6.2 RBCC — An Example Function . . . 52

6.2.1 The Allocated Bandwidth Table . . . 53
6.2.2 Deciding to Update the ABTs . . . 54
6.2.3 Updating the ABTs . . . 55
6.2.4 Updating the Packet's Congestion Fields . . . 57
6.3 TRUMP/RBCC Operation . . . 57
6.3.1 Source Transmission . . . 57
6.3.2 Packet Reception by the Destination . . . 58
6.3.3 Packet Reception by the Source . . . 59
6.3.4 Flow Termination . . . 59
6.3.5 Abnormal Loss of a Traffic Flow . . . 59
6.3.6 Congestion Avoidance . . . 60
6.4 Summary . . . 60

7 Comments on the Congestion Control Framework 62
7.1 The Framework as a Feedback System . . . 62
7.2 What Constitutes a Traffic Flow? . . . 63
7.3 What Congestion Fields are Required? . . . 64
7.4 Transport and Network Layer Intercommunication . . . 65
7.5 Complexity of RBCC . . . 66
7.6 RBCC and Fair Share . . . 67
7.7 Determination of Desired Rate . . . 68
7.8 Bandwidth Reservation and Resource Monitoring . . . 69
7.9 Acknowledgment Flow Problems . . . 69
7.10 Packet Admission and Readmission . . . 70
7.11 Summary . . . 70

8 Implementing TRUMP/RBCC in the REAL Network Simulator 72
8.1 The REAL Simulator . . . 72
8.2 Changes to the Simulator . . . 73
8.3 Implementation of TRUMP . . . 74
8.3.1 TRUMP Destination Code . . . 75
8.3.2 TRUMP Source Code . . . 75
8.4 Implementation of RBCC . . . 77
8.5 Simulation of TCP in REAL . . . 79
8.6 Measurements Obtained by The REAL Simulator . . . 80

8.6.1 Indicators of Network Congestion . . . 80
8.6.2 Other Indicators of Network Performance . . . 81
8.7 Summary . . . 81

9 Results of Simulating TRUMP and RBCC in REAL 82
9.1 Introduction . . . 82
9.2 Description of Scenarios . . . 82
9.2.1 Scenario Variations . . . 83
9.2.2 Tabled Results . . . 84
9.3 Scenario 1 . . . 84
9.3.1 Transmission Sequence Numbers . . . 84
9.3.2 Router Queue Lengths . . . 85
9.3.3 Round-Trip Times . . . 86
9.4 Scenario 2 . . . 86
9.4.1 Optimum Rates for Scenario 2 . . . 87
9.4.2 Transmission Sequence Numbers . . . 87
9.4.3 Packet Loss . . . 88
9.4.4 Router Queue Lengths . . . 89
9.4.5 End-to-End Delays . . . 90
9.4.6 Link Utilisations . . . 91
9.4.7 Predicted and Measured TRUMP Transmission Rates . . . 92
9.4.8 Evaluation of Comparison Results . . . 92
9.5 Scenario 2a . . . 92
9.6 Scenario 3 . . . 93
9.6.1 Transmission Sequence Numbers . . . 94
9.6.2 Router Queue Lengths . . . 94
9.6.3 Link Utilisations . . . 95
9.6.4 End-to-End & Round-trip Times . . . 96
9.6.5 TRUMP Transmission Rates . . . 97
9.7 Scenario 4 . . . 97
9.8 Scenario 5 . . . 100
9.9 Scenario 6 . . . 104
9.10 Scenario 7 . . . 105
9.11 Scenario 8 . . . 107
9.12 Scenario 9 . . . 109

9.13 Summary . . . 111

10 Simulating Random Scenarios **112**

10.1 Introduction . . . 112
10.2 Scenario Generation . . . 112
10.3 Abstracted Measurements . . . 113
10.4 Highest Average Buffer Queue Length . . . 113
10.5 Average Buffer Queue Length . . . 114
10.6 Total Packets Lost . . . 114
10.7 Highest Average Link Utilisation . . . 116
10.8 Effective Bit Rate . . . 117
10.9 End-to-end Variance . . . 118
10.10 Inter-packet Spacing . . . 118
10.11 ABT Caching Results . . . 119
10.12 Summary of Results . . . 120

11 Effect of Parameters on Framework Performance **121**

11.1 Available Parameters . . . 121
11.2 Measurements Made . . . 122
11.3 Effect of Rate Quench Packets . . . 122
11.4 Effect of Reverse ABT Updates . . . 123
11.5 Effect of the Threshold, T . . . 124
11.6 Effect of the Scaling Factor, α . . . 126
11.7 Effect of the Selective Acknowledgment Size . . . 127
11.8 Effect of the Packet Dropping Function . . . 128
11.9 Conclusion . . . 129
11.10 Results with Recommended Parameters . . . 130

12 Conclusion **131**

12.1 Introduction . . . 131
12.2 A New Rate-Based Control Framework . . . 132
12.3 Framework Functions . . . 133
12.4 Experimental Results . . . 134
12.5 Future Work . . . 135
12.6 Summary . . . 137

A Glossary **138**

B Verifying TRUMP with Spin **142**

B.1 Introduction to Spin . . . 142

B.2 The Promela Language . . . 143

B.2.1 Executability . . . 143

B.2.2 Data Types . . . 143

B.2.3 Processes . . . 144

B.2.4 Atomic Sequences . . . 144

B.2.5 Message Channels . . . 144

B.2.6 Control Flow . . . 145

B.2.7 Promela Verification Support . . . 146

B.3 Spin Verification Capabilities . . . 146

B.4 Implementing TRUMP in Promela . . . 147

B.4.1 TRUMP's Promela Implementation . . . 148

B.5 TRUMP State Diagrams . . . 161

B.5.1 TRUMP Sender State Diagram . . . 161

B.5.2 TRUMP Receiver State Diagram . . . 162

B.6 Verification of TRUMP . . . 163

B.6.1 No Features Enabled . . . 164

B.6.2 Selective Acknowledgments . . . 164

B.6.3 Receiver Errors . . . 164

B.6.4 Implicit Connection Setup . . . 165

B.6.5 Packet Losses . . . 165

B.6.6 Packet Reordering . . . 166

B.7 Summary . . . 166

C TRUMP Protocol Headers **167**

C.1 Connection Setup . . . 169

C.2 Special Acknowledgment . . . 169

C.3 Data Transmission . . . 170

C.4 Data Acknowledgment . . . 171

C.5 Connection Termination . . . 172

D Complexity of RBCC **173**

D.1 Introduction . . . 173

D.2 Estimation of Flows through a Router . . . 173
D.3 Size of RBCC Data Structures . . . 174
D.4 Packet Arrival for an Existing Flow . . . 175
D.5 Packet Arrival for a New Flow . . . 178
D.6 Removal of an Existing Flow . . . 178
D.7 Route Changes . . . 178
D.8 Summary . . . 179

E REAL NetLanguage Files for Simulated Scenarios 180
E.1 Scenario 1 . . . 181
E.2 Scenario 2 . . . 181
E.3 Scenario 3 . . . 182
E.4 Scenario 4 . . . 183
E.5 Scenario 5 . . . 184
E.6 Scenario 6 . . . 185
E.7 Scenario 7 . . . 186
E.8 Scenario 8 . . . 187

F Theoretical Rate Calculation for Scenario 2 189

G Obtaining the REAL Simulator 191

Chapter 1

Introduction

The control of congestion in connectionless packet-switched wide-area networks is a major problem, especially in such networks as the Internet, which is experiencing an exponential growth in users and network traffic. This thesis outlines a rate-based framework for congestion control in these networks, examines the requirements of the framework, and describes a number of control mechanisms which meet the framework's requirements.

The first section (Chapters 1 to 3) describes the problem of congestion in these networks, and reviews the current methods employed to deal with congestion, plus other solutions that are described in the literature. The second section (Chapters 4 to 7) outlines my proposed rated-based congestion control framework, and describes its ramifications and parameters. Finally, the framework is tested in several network simulations where it is compared against some existing methods, and the third section (Chapters 8 to 11) gives experimental results and summarises them.

1.1 Connectionless Packet-switched Networks

Most computer networks suffer from network congestion to some extent. This thesis limits its scope to congestion in *connectionless packet-switched wide-area networks*, and in particular, to networks with the same architecture as the Internet.

The network architecture under consideration has the following characteristics:

1. The network is a *wide-area network* with nodes in the network passing data to other nodes along links. The nodes are *arbitrarily* connected; the network does not have a particular topology (e.g spanning tree, hypercube etc.).
2. The network is *connectionless*. There is no reservation of network bandwidth or resources between a source of data transmission and its intended destination. End-to-end "connections" may be set up by the Transport Layers in the source and destination, but the other nodes within the network are oblivious to these Transport Layer operations.
3. The network is *packet-switched*. Packets are individually routed from source to destination. They may take different routes, and may arrive at the destination out-of-order.
4. Variable-sized packets are routed from one link to the next via nodes, often known as *packet switches*, *gateways*, or more commonly, *routers*. Routers usually buffer incoming packets

before they are transmitted on an outgoing link. This is described in greater detail in the next section.

5. The network uses *semi-static routing*. Routes can change slowly over time, but most routes are static.
6. The network sets *no bandwidth constraints* on data sources. A source host may attempt to transmit data at a rate which will exceed the bandwidth or resources on the links and nodes between it and the destination host.
7. Links are assumed to have fixed bandwidths with no service guarantees. There is no expectation of error- or flow-control on any link.
8. The network makes *no transmission guarantee*. The links on the network have finite bandwidth, and the nodes have finite buffer space for packets waiting to be routed. If network components are unable to forward data on to the destination — for any reason — the data may be discarded. This type of network is also known as a *best effort* network.
9. The network is an *internetwork*. The nodes in the network are connected by links that vary greatly in type. There may be large variations in:

    - The physical characteristics of the links,
    - The Medium Access Control mechanism of the links,
    - The bit rate on the links,
    - The bit error rate on the links, and
    - The maximum packet size on the links.

10. The bandwidth on each link is generally static. It usually changes only when upgrades to the network infrastructure are performed.

1.2 Router Operation

All the links on a network are joined together by routers. These forward packets arriving on incoming links to the appropriate outgoing links, to ensure that the packet is routed to its intended destination. Figure 1 shows the basic architecture of a router.

The router is connected to I incoming links and O outgoing links. I and O may be different, although I usually equals O. In most situations, input and output links are paired to form either a full-duplex channel where data can flow in both directions simultaneously, or a half-duplex channel where data can flow in only one direction at a time.

Incoming packets are buffered in the input buffers. Once in the buffer, they are selected by the Packet Selection Function to be passed to the Routing Function. The Routing Function determines on to which outgoing link a packet must be placed for it to get closer to its destination: this is done with a routing table which, as described earlier, is semi-static. When the correct link for the packet is found, the Routing Function passes the packet to the Packet Dropping Function for queueing on the output buffer for the link. When the packet reaches the other end of the queue, it is transmitted over the link to the next router, or to the final destination.

The Packet Selection Function can choose any of the packets queued in the input buffers for routing by the Routing Function. Normally this would be done in a First-In First-Out method, but other selection methods may be useful in a congested environment.

There are two fundamental bottlenecks in the given router architecture. Firstly, there is a minimum time needed for the router to decode the network header of the incoming

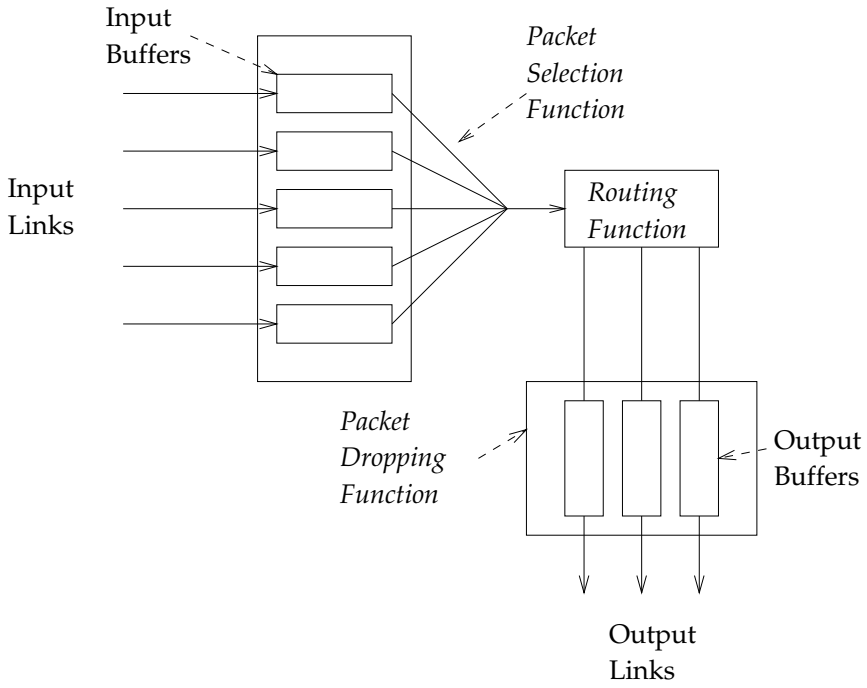


Figure 1: Basic Router Architecture

packet, determine the route for the packet, and pass the packet to an outgoing link for transmission. There is also a delay for the packet to be transmitted on the outgoing link: this may just be the packet's transmission time for a full-duplex link, or there may be an extra delay for the link to become clear on a half-duplex link. These delays form one bottleneck.

The second bottleneck indicates that the router must be prepared to buffer output packets, to prevent them from being lost while the router waits for an outgoing link to become clear. The two bottlenecks together indicate that the router must buffer incoming packets, to prevent them from being lost if they arrive too quickly for the router to process.

By definition, the router's input and output buffers are finite. If a buffer becomes full, then no more packets can be queued in the buffer, and the router has no choice but to discard them. This causes data loss in the data flow between a source and destination, and usually causes the source to retransmit the data.

Although a router cannot queue a packet if the corresponding output buffer is full, it can choose either to discard the unqueued packet, or to discard a packet already in the output queue, and then queue the unqueued packet. This choice is performed by the Packet Dropping Function. This cannot be done for the input buffers, as the packet has not been placed in the router's internal memory until it has been queued in the input buffers. Thus, packets may be lost on input, and the router has no control over which packets are lost.

Finally, the router's buffers may share a common memory space, or they may have individual memory spaces. With the former, no buffer can become full until the router's entire memory space is allocated to buffers, at which point *all* buffers become full. With the latter, any buffer's utilisation has no influence on any other buffer's utilisation.

1.3 Congestion in Connectionless Packet-switched Networks

A network is congested when one or more network components must discard packets due to lack of buffer space. Given the above architecture, it is possible to see how network congestion can occur. A source of data flow on the network cannot reserve bandwidth across the network to its data's destination. It, therefore, is unable to determine what rate of data flow can be sustained between it and the destination.

If a source transmits data at a rate too high to be sustained between it and the destination, one or more routers will begin to queue the packets in their buffers. If the queueing continues, the buffers will become full and packets from the source will be discarded, causing data loss. If the source is attempting to guarantee transmission reliability, retransmission of data and increased transmission time between the source and the destination is the result. Figure 2 from [Jain & Ramakrishnan 88] demonstrates the problem of congestion.

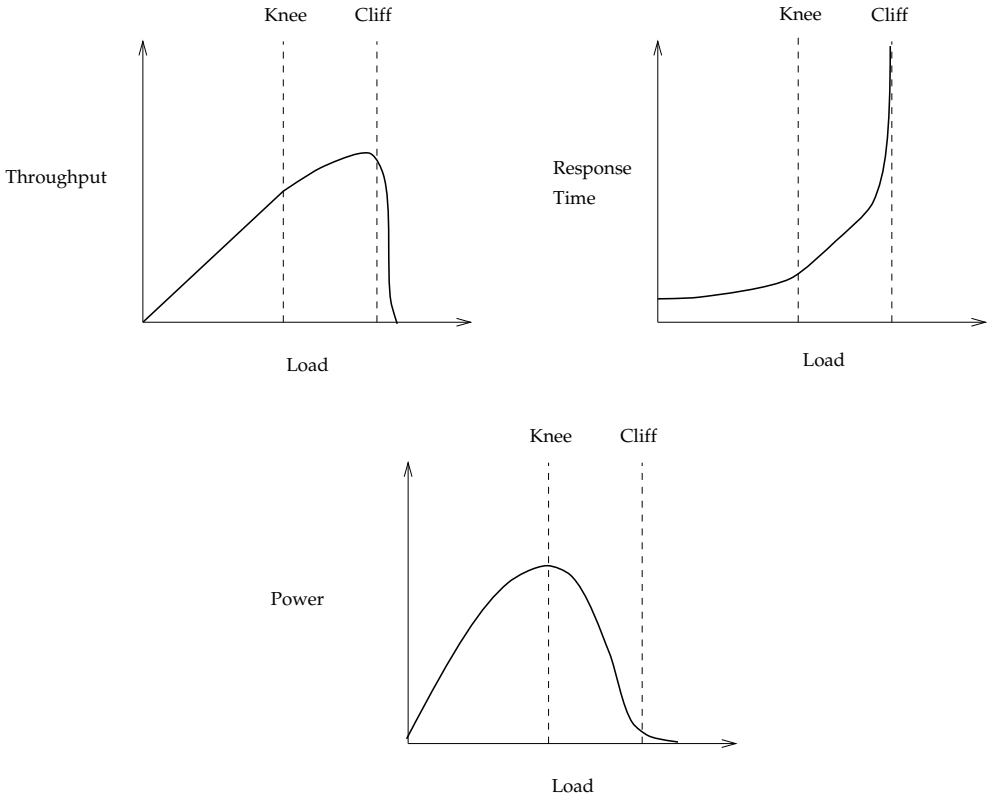


Figure 2: Network Performance as a Function of Load [Jain & Ramakrishnan 88]

As the load (rate of data transmitted) through the network increases, the throughput (rate of data reaching the destination) increases linearly. However, as the load reaches the network's capacity, the buffers in the routers begin to fill. This increases the response time (time for data to traverse the network between source and destination) and lowers the throughput.

Once the routers' buffers begin to overflow, packet loss occurs. Increases in load beyond this point increase the probability of packet loss. Under extreme load, response time approaches infinity and the throughput approaches zero; this is the *point of congestion collapse*. This point is known as the *cliff* due to the extreme drop in throughput. Figure 2 also shows a plot of power, defined as the ratio of throughput to response time. The power peaks at the *knee* of the figure.

1.4 Congestion Control Schemes

Network congestion control schemes can be divided into two categories. *Congestion avoidance* schemes attempt to keep the network operating at the knee of Figure 2. *Congestion recovery* schemes attempt to keep the network operating to the left of the cliff in Figure 2. A congestion avoidance scheme attempts to keep the load on the network to a level where response time is low, and power is optimised. A congestion recovery scheme attempts to keep the network operating by preventing the loss of data due to congestion.

Jain notes in [Jain & Ramakrishnan 88] that "Without congestion recovery a network may cease operating, whereas networks have been operating without congestion avoidance for a long time." Clearly, although congestion recovery prevents network collapse, it is better to operate the network at the point where power is maximised.

1.5 Factors Affecting Network Congestion

Network congestion is affected by the following factors:

1. The amount of traffic generated and placed on the network by its users;
2. The network architecture in terms of its links, their connectivity, and the characteristics of each link;
3. Operating parameters of sources, routers and destinations — such as available buffers, available processing power and system design;
4. Flow control mechanisms used in the Transport Layers of sources of data flow and their respective destinations;
5. Congestion avoidance and congestion recovery schemes in routers and sources;
6. Packet admission mechanisms in the Network and Link layers of sources and destinations;
7. Packet discarding and packet loss mechanisms in routers and destinations.

1.6 Scope of This Thesis

The scope of this thesis is constrained to the protocols and procedures used in the Transport and Network Layers of packet sources, routers and packet destinations of connectionless packet-switched wide-area networks. Therefore, the first three factors fall outside the scope of this thesis, and I will give them little or no coverage.

The next two chapters will review network congestion control schemes which have either been proposed or implemented. Following the thesis scope, only techniques operating at the Transport and Network Layers will be considered. The advantages and disadvantages of each mechanism will be discussed.

Given the deficiencies I will identify with many existing congestion control schemes, and the advantages of rated-based controls schemes that I will highlight, I will then present a rate-based congestion control framework which is designed for connectionless packet-switched networks.

Chapter 2

Existing Congestion Control Schemes

There are many existing congestion control schemes for connectionless packet-switched networks. In this chapter, I will review the topic of congestion control in more detail, and examine some traditional control schemes. In the next chapter, I will focus on congestion control schemes which are rate-based, and which are more closely related to the work described in this thesis.

2.1 Jain's Congestion Survey

Jain's review paper [Jain 90] surveys the problems and possible solutions of network congestion control. Jain cites several myths about congestion:

- Congestion is caused by a shortage of buffer space, and can be solved by increasing the amount of buffers.
- Congestion is caused by slow links.
- Congestion is caused by slow processors.
- If not one, then solutions to all of the above will cause the congestion problem to go away.

As shown in [Nagle 87], congestion cannot be solved with a large buffer: this only postpones the inevitable packet loss when network load is larger than network capacity. Slow links by themselves do not cause congestion, but the mismatch of network link speeds helps cause congestion. Similarly, slow processors do not cause congestion: the introduction of a high-speed processor in an existing network may actually increase the mismatch of processing speeds and the chances of congestion. In fact, congestion occurs even if all links and processors are of the same speed.

Jain suggests that there are two main classes of congestion control schemes: *resource creation* schemes, which dynamically increase the capacity of the resource, and *demand reduction* schemes, which reduce the demand on the resource. For the networks under consideration, the bandwidth on each link is generally static; therefore, we must discount resource creation schemes. Demand reduction schemes can be subdivided into three subclasses:

- *Service denial* schemes do not allow further resource allocation. Connection-oriented networks, for example, can prevent new connections from being formed.
- *Service degradation* schemes ask all users (existing as well as new) to reduce their loads.
- *Scheduling* schemes ask users to schedule their demands so that the total demand is less than the capacity. This subclass is a special instance of the service degradation subclass.

In connectionless networks, new network transmissions cannot be prevented, so a service denial approach cannot be used. For these networks, we must consider scheduling and service degradation schemes.

Several feedback techniques have been proposed to implement these approaches [Jain 90], such as:

- *Observing congestion evidence* such as packet loss or increasing packet delays, and reducing the load on the network where possible [Jain 86] [Jacobson 88]. This is done primarily at the Transport Layer and not within the network itself. One drawback with this approach is that packet loss and increasing delays do not always indicate congestion.
- *Feedback messages* which are sent from the congested resource to the source point. Examples are "choke packets" [Majithia & *et al* 79], "quench messages" [Postel 81a] and "permits" [Davis 72]. A drawback with these approaches is that the extra traffic created can exacerbate the congestion.
- *Feedback in routing messages* allows an intermediate resource to set its load level to that of all neighbouring nodes.
- *Rejecting further traffic* causes incoming packets to be lost or not acknowledged. The resulting *backpressure* causes queues to form at other nodes, and the backpressure slowly travels towards the source. This technique is only useful for short duration congestion, and can affect other resources not involved in the congestion.
- *Probe packets* are sent through the network, and source nodes adjust their loads, according to the delays experienced by the probe packets.
- *Feedback fields in packets* can pass congestion information to end hosts [Schwartz 82], [Robinson *et al* 90]. This avoids the overhead of feedback messages, and can be used in both forward and reverse directions.

Also, several alternatives has been proposed for the location of congestion control, such as in the Transport Layer, Network Layer, Data Link Layer, and in specific network entrances, such as routers.

There are several requirements for a good congestion control scheme. The scheme must have a low overhead. The scheme must be fair: defining fairness is not trivial, and no one definition has been widely accepted. The scheme must be responsive to available capacity and demand. The scheme must work in the reduced environment of congestion, including effects such as transmission errors, out-of-sequence packets and lost packets. Finally, the scheme must be socially optimal: it must allow the total network performance to be maximised. Designing a congestion scheme that meets all of these requirements is not trivial.

In conclusion, Jain restates that congestion is not a static resource shortage problem, but a dynamic allocation problem. Congestion control schemes can be classified as resource creation schemes or demand reduction schemes. Congestion control is not trivial because of the number of requirements such as low overhead, fairness etc. A number of network policies affect the

choice of congestion control schemes, and one congestion control scheme may not be suitable for all networks. Finally, as networks become larger and heterogeneous, with higher speeds and integrated traffic, congestion will become more and more difficult to handle, but also more and more important to control.

2.2 Traditional Control Schemes

In most traditional connectionless packet-switched networks, the functionality of the Network Layer and Transport Layer are clearly separated [Clark 88]. The Network Layer is responsible for the forwarding of packets to their destination host, across one or more links which may have widely different transmission characteristics: total bit rate, overall bit error rate, medium access method, multiplexing scheme etc. In the networks under consideration, this delivery of packets is "best effort", with no guarantee of packet delivery: intermediate routers may discard packets for any reason. The Network Layer perceives *flows* of traffic from source hosts to destination hosts.

The Transport Layer is responsible for such things as the setup of virtual circuits (also known as *connections*) between a source application and a destination application, the reliable and in-sequence delivery of data to the destination application, and flow & congestion control. Not all of these value-added services are required by every application, and some applications may desire other services such as constant-bit-rate connections.

In these connectionless packet-switched networks, congestion control schemes are generally implemented in the Transport Layer of the source and destination hosts. The Transport Layer must counter packet loss, which is often caused by network congestion. As well, an existing Transport Layer flow control scheme can often be adapted to perform some congestion control.

The Transport Layer, as the basic generator of a traffic flow, is the essential "source" of the traffic flow. Unfortunately, while it is responsible for the flow of traffic that may cause network congestion, the Transport Layer is divorced from the Network Layer. Therefore, the Transport Layer is unable to see the effect of its traffic flow on the intermediate nodes between it and the destination of its traffic.

The Transport Layer is only able to determine that there is network congestion in two ways:

- By being informed of existing network congestion from the Network Layer (as is done in the Source Quench mechanism described in Section 2.4.1).
- By observing the effects of congestion on the traffic flow: delayed arrival of acknowledgments, and acknowledgments that indicate the loss of data (as is described in [Jacobson 88]).

Delayed acknowledgments and loss of data do **not** specifically imply network congestion. They may be caused by packet corruption and temporary traffic instability due to new flows of data being created. Therefore, transport protocols that rely on the second method to deduce the existence of congestion may not detect congestion when it exists, and may believe that congestion exists when it doesn't.

Transport protocols which perform end-to-end reliability must retransmit data when it is lost in transit. In general, retransmissions are prompted either by the arrival of a negative acknowledgment, or the expiry of a timer set to detect the non-arrival of any acknowledgment. Neither of these events specifically imply the loss of data. Transport protocols which use these events

to prompt data retransmission may retransmit data unnecessarily, burdening the network with redundant transmissions and increasing the chance of congestion.

Many existing transport protocols suffer from the problems described above. The transport protocol which typifies the Transport Layer in these traditional networks, and which has seen the most research effort in its congestion control, is TCP. Let us examine this work.

2.3 Transmission Control Protocol

The Transmission Control Protocol, or TCP, provides reliable end-to-end data transmission across a possibly unreliable network such as the Internet. TCP was influenced by the NCP protocol [Reynolds & Postel 87] and by [Cerf & Kahn 74]. Originally standardised in [Postel 81b], it has since been improved upon by several researchers.

2.3.1 Sliding Window Flow Control

TCP uses a dynamic Go-Back-N sliding window for end-to-end flow control. The size of the window is negotiated during the connection setup, and can vary during the course of data transmission. The receiver returns an 'Advertised' window size in every acknowledgment which indicates the amount of buffer space the receiver has left for this traffic flow. The sender calculates an 'Effective Window' size, which is how much more traffic it can transmit to the receiver. The effective window size is equal to the receiver's advertised window size minus the amount of data currently outstanding. If the effective window size is less than one byte, then the sender cannot transmit any further data.

The sliding window in TCP also affects its retransmission strategy. If an acknowledgment from the destination shows that some data beginning at point N is lost, the source may need to retransmit **all** outstanding data from N onwards, even if the destination has already received much of this data. This causes unnecessary network traffic and increases the network load.

Since the original specification of TCP was published, its flow control has been found to be fraught with several problems and deficiencies. There have been several proposals that remedy these problems, such as the "Silly Window Syndrome" [Clark 82] and "Small Packet Avoidance" [Nagle 84]. Other remedies address the use of the sliding window for congestion control, and are described below.

Nearly all implementations of TCP have these sliding window extensions. TCP's sliding window operation is now well-behaved but very inelegant.

2.3.2 Sliding Window Congestion Control

The initial TCP specification allowed hosts to transmit as much data as the Effective Window size, to fill the destination's buffers. This, however, could cause significant congestion in intermediate routers, and indeed was one of the main causes for the high Internet congestion in the mid 1980s [Nagle 84] [Jacobson 88].

Several modifications to the sliding window scheme were proposed in [Jacobson 88] which dramatically reduced the problem of network congestion. These were "Multiplicative Decrease", "Slow Start" and "Fast Retransmit".

If the assumption that packet loss generally indicates network congestion is true, then this

also indicates that the source has too much in-transit data. The Multiplicative Decrease proposal suggests:

1. Add a variable *cwnd* (the congestion window) to the sender's state.
2. When packet loss is detected, set *cwnd* to half the current window size (the multiplicative decrease).
3. On each acknowledgment for successful data reception, increase *cwnd* by $1/cwnd$.
4. At any time, transmit packets so that the amount of in-transit data is the minimum of the Effective Window and *cwnd*.

With this scheme, TCP slowly increases its congestion window size until data is lost, then the congestion window is halved. This helps to drain excess packets from the network. This pattern of increase and decrease continues until an equilibrium is reached. Changes within the network itself may disturb the equilibrium, but the Multiplicative Decrease scheme helps to return data transmission to a near-equilibrium point.

The Multiplicative Decrease scheme does not address how equilibrium is initially reached, only what to do when this has occurred. Reaching equilibrium is covered by Slow Start:

1. When starting data transmission, or when the Effective Window size in the source is zero, set *cwnd* to one packet.
2. On each acknowledgment for successful data reception, increase *cwnd* by one packet.
3. At any time, transmit packets so that the amount of in-transit data is the minimum of the Effective Window and *cwnd*.
4. Revert to Multiplicative Decrease on packet loss.

Slow Start causes an exponential increase in the congestion window until data is lost. Initially, *cwnd* is one packet. When the ack for this packet arrives, *cwnd* becomes two and two new packets are transmitted. When their acks arrive, *cwnd* is incremented twice to become four, and so on. Eventually, the congestion window becomes so large that packet loss occurs, and Multiplicative Decrease takes over.

Slow Start is also used when the source cannot transmit because the effective window size is zero. If this window of outstanding data is acknowledged all at once by the destination, then the source could retransmit a whole advertised window of data, which may cause short-term network congestion. Instead, Slow Start is used to ramp the source's window of outstanding data up to this advertised size.

These two schemes have been very effective in reducing the network congestion caused by TCP's sliding window. However, they depend upon the accurate detection of packet loss in order to operate correctly. TCP's use of timers to detect packet loss is covered in the next section. Another scheme to detect packet loss, "Fast Retransmit" was also proposed in [Jacobson 88].

Even though TCP sets a timer to detect if a packet is lost (as no acknowledgment arrives), acknowledgments for other data may indicate the loss of a packet. This is because the Go-Back-N acknowledgment scheme used by TCP only acknowledges the highest point of *contiguous* data received to date.

With Fast Retransmit, data is retransmitted once three or more acknowledgments with the same sequence number are received by the source, even if the packet loss timer has not yet expired. Two acknowledgments with the same sequence number may have been a consequence of data reordering within the network. Fast Retransmit helps TCP to react to packet loss faster than with just the packet loss timer.

2.3.3 Retransmission Timer Deficiencies

TCP normally uses the expiry of a timer to detect the non-arrival of an acknowledgment: this in turn prompts data retransmission. The dilemma here is that if the timer is set too small, it expires before an acknowledgment can arrive, and data is retransmitted unnecessarily. If it is set too large, the protocol reacts too slowly to the loss of data, reducing the rate of data transmission and throughput.

The acknowledgment for transmitted data should return at the round-trip time. However, the round-trip time is anything but constant, and it varies due to the load on the network and short- and long-term congestion within the network. Thus, estimating the round-trip time is very difficult. The original specification for TCP [Postel 81b] suggested a simple algorithm for estimating the round-trip time.

Both Zhang [Zhang 86] and Jacobson [Jacobson 88] demonstrate that the algorithm does not cope with the large round trip time variance often experienced when the Internet is heavily loaded. Jacobson describes a method for estimating the variance in the round-trip time. This method causes fewer unneeded TCP timer timeouts, reducing unneeded packet retransmissions and reducing the load and the congestion in the network. All major implementations of TCP now have Jacobson's round-trip estimation algorithm in their TCP code.

Karn [Karn & Partridge 87] added to Jacobson's work by showing an ambiguity in round-trip time estimation due to acknowledgments for packets that have been retransmitted. Karn presents an algorithm for removing the ambiguity from round-trip time estimates. Again, all major implementations of TCP now have Karn's algorithm in their TCP code.

These solutions show that round-trip time estimation is difficult to do in conditions when it has large variance. Incorrect round-trip time estimation leads to network congestion and/or poor throughput and can further add to round-trip time variance, compounding the problem.

2.3.4 TCP Vegas

Despite the improvements described above, TCP only implements congestion *recovery*, as it continually increases its congestion window until packets are lost. This probing of the network increases buffer occupancy in routers, shifting network operation past the knee-point of peak power (Figure 2, pg. 4).

Most of the improvements in [Jacobson 88] were implemented in the 4.3BSD-Tahoe operating system, and this version of TCP is known as TCP Tahoe. The 4.3BSD-Reno operating system added Fast Recovery, TCP header prediction and delayed acks to TCP, and this version of TCP is therefore known as TCP Reno.

Since the implementation of TCP Reno, several researchers have highlighted shortcomings in its implementation and in its congestion recovery scheme [Wang & Crowcroft 81] [Wang & Crowcroft 91] [Brakmo & Peterson 95]. Recently, further proposed modifications to TCP, known as TCP Vegas [Brakmo *et al* 94], suggest several enhancements to TCP Reno. Their research also describes a congestion *avoidance* scheme which allows TCP to find an equilibrium in transmission, without inducing network congestion to do so.

In TCP Reno, the Fast Retransmit scheme depends on at least three duplicate acks, which is unlikely in high congestion situations where there is large packet loss. TCP Vegas records the system time for each packet transmitted; therefore, it can determine the round-trip time accurately for each acknowledgment. If a duplicate acknowledgment has a round-trip time greater than the time set for the data's packet loss timer, then the packet is retransmitted. Three or more duplicate acknowledgments are not required. TCP Vegas also retransmits data before the expiry of the packet loss timer for other reasons.

In this modified Fast Retransmit scheme, the congestion window is never reduced when data is retransmitted because it was lost *before* the congestion window was previously reduced. This was possible in TCP Reno and would cause lower throughput.

The congestion avoidance scheme in TCP Vegas, which replaces Multiplicative Decrease, depends on the idea that the congestion window is directly proportional to the throughput expected by TCP; the congestion window should only be increased if the throughput expected by TCP is increasing. TCP Vegas approximates a measurement of the expected throughout for the connection, and uses this to update the congestion window, as described in [Brakmo *et al* 94]

Results given in [Brakmo *et al* 94] and [Ahn *et al* 95], from simulations and Internet experiments, indicate that TCP Vegas gives greater throughput for less packet loss than TCP Reno. TCP Vegas was also found to keep buffer occupancy in routers at a lower level than TCP Reno. These results are confirmed by the experimental results described in this thesis.

Although TCP Vegas appears to provide very good congestion control, several researchers[1] believe that TCP Vegas is too aggressive in its use of bandwidth, penalising TCP Reno sources and not allowing new TCP connections to obtain a fair share of the available bandwidth. Other papers suggest deficiencies in TCP Vegas [Ait-Hellal & Altman 96]. The verdict on TCP Vegas is still out.

2.3.5 Summary

TCP as first specified was a complicated transport protocol, and the improvements to it since then has made it even more complicated. The most widely-used implementation, TCP Reno, only has congestion recovery. Even worse, TCP must cause network congestion to determine its congestion window size, and its reliance on round-trip estimation ensures that both error recovery and congestion recovery are not optimal. Suggested congestion avoidance schemes such as TCP Vegas have not yet been fully accepted by the networking community.

Chapters 9 and 10 highlight the congestion deficiencies in TCP Reno, and also show the benefits in congestion control that TCP Vegas affords.

[1] Sally Floyd, Van Jacobson and others, End-to-end mailing list *end2end-tf@isi.edu*, 1994.

2.4 Network-based Congestion Control

Congestion control schemes where network components such as routers participate should be more effective than purely end-to-end control schemes. As most network congestion occurs in routers, they are in an ideal position to monitor network load and congestion, and use this information in a congestion control scheme.

Network-based congestion control schemes can feedback congestion data to transmitting sources, thus altering their behaviour directly. Alternatively, network-based schemes can change the packet processing within routers: this can help alleviate congestion, and may also indirectly affect sources' behaviour.

We will examine several schemes that feedback congestion data to sources, and several schemes that alter packet processing behaviour within routers.

2.4.1 ICMP Source Quench

The ICMP Source Quench is the only congestion control mechanism in the Network Layer of the TCP/IP protocol suite. Both routers and hosts play a part in the mechanism to control congestion. When a router believes itself to be congested, it sends 'Source Quench' packets back to the source of the packets causing the congestion. On receipt of these packets, the host should throttle its data rate so as to prevent router congestion. It can then slowly increase its data rate, until source quench packets begin to arrive again.

The mechanism is simple, and suffers from many deficiencies in both the effectiveness of the mechanism, and the specification of its implementation. Let us tackle the latter first.

The Source Quench mechanism is defined in [Postel 81a]. In effect, the specification of the Source Quench mechanism makes **no** proscriptions with regards its implementation in any Internet protocol stack. The entire Source Quench mechanism is thus left to each implementation to decide on its particular behaviour. This may lead to behaviour that is different to the spirit of the specification, and possibly to badly-behaved congestion control between two implementations of the same mechanism.

Source Quench's effectiveness as a congestion control mechanism is also flawed. This is chiefly due to two problems. Firstly, [Postel 81a] does not clearly state how a router or destination decides when it is congested, who to send a source quench message to, how often to send source quench messages, and when to stop. Secondly, [Postel 81a] does not clearly state how a host should react to a source quench message, by how much it should reduce its output rate, if it should inform upper layers in the network stack, and how to properly increase its output rate in the absence of source quench messages.

Even worse, a router or destination may be congested and not send any source quench messages. Sources in receipt of a source quench message cannot determine if a router or destination dropped the datagram that caused the source quench message. The source also does not know to what rate it should reduce its transmission to prevent further source quench messages.

Since the original specification of the Source Quench mechanism, several further papers have described alternatives to improve the performance of the mechanism, and to tighten the mechanism's specification ([Nagle 84] and [Braden & Postel 87]). One serious problem, raised by Nagle in [Nagle 84], is that sources who receive source quench messages may not inform the corresponding Transport Layer of the source quench. Most implementations only inform TCP, leav-

ing other protocols such as UDP [Postel 80] unaware that they are causing congestion. This is a serious problem, as the indication of a congestion problem is not communicated to the layer that is generating the packets that are causing the problem.

Some of the deficiencies in Source Quench were recognised in [Braden & Postel 87], and while it mandated certain parts of the Source Quench mechanism, it still left others non-mandatory, with suggested behaviour given. The document notes that "Although the Source Quench mechanism is known to be an imperfect means for Internet congestion control, and research towards more effective means is in progress, Source Quench is considered to be too valuable to omit from production routers."

Source Quench was reviewed in [Mankin & Ramakrishnan 91]. The review is small, and is given here in summary, with editorial emendations.

> The method of router congestion control currently used in the Internet is the Source Quench message. When a router responds to congestion by dropping datagrams, it may send an ICMP Source Quench message to the source of the dropped datagram. This is a congestion recovery policy.
>
> [...]
>
> [A] significant drawback of the Source Quench policy is that its details are discretionary, or, alternatively, that the policy is really a family of varied policies. Major Internet router manufacturers have implemented a variety of source quench frequencies. It is impossible for the end-system user on receiving a Source Quench to be certain of the circumstances in which it was issued. [...]
>
> To the extent that routers drop the last arrived datagram on overload, Source Quench messages may be distributed unfairly. This is because the position at the end of the queue may be unfairly often occupied by the packets of low demand, intermittent users, since these do not send regular bursts of packets that can preempt most of the queue space.

In summary, it can be seen that Source Quench is a rudimentary congestion control method. It is loosely specified, which can lead to badly-behaved operation across different implementations. It may not limit a source's transmission rate, depending on the transport protocol, and it may drop packets unfairly.

Despite this, Source Quench is in wide operation throughout the Internet, and provides the sole network-level congestion control for it.

2.4.2 The Dec Bit Mechanism

The Dec Bit mechanism, also known as the Congestion Indication mechanism, is a *binary feedback* congestion avoidance mechanism developed for the Digital Network Architecture at DEC [Jain *et al* 87], and has since been specified as the congestion avoidance mechanism for the ISO TP4 and CLNP transport and network protocols.

In Dec Bit, all network packets have a single bit, the 'Congestion Experienced Bit', in their headers. Sources set this bit to zero. If the packet passes through a router that believes itself to be congested, it sets the bit to a one. Acknowledgment packets from the destination return the received Congestion Experienced Bit to the source. If the bit is set, the source knows there was

some congestion along the path to the destination, and takes remedial action. In DECNET, the source adjusts its window size. In TP4, the destination alters the advertised window size, rather than returning the bit to the source.

Dec Bit can be used as either a congestion avoidance or a congestion recovery mechanism, but is used as an avoidance mechanism in [Jain *et al* 87]. To be used as an avoidance mechanism, routers must determine if their load is at, below or above their peak power point (see Figure 2, pg. 4). This is achieved by averaging the size of the router's output queues. If they have more than one buffered packet on average, that output interface's load is above the peak power point; if less than one packet, the interface's load is below the peak power point.

As noted by Jain et al. [Jain *et al* 87], determining a good averaging function is not easy. Using instantaneous queue sizes is an inaccurate indication of the load on an interface. On the other hand, a too-large averaging interval gives a distorted indication of the real load. They settled for an average taken from the second-last 'regeneration point' to the current time, where a 'regeneration point' is a point when the queue size is zero, as shown in the following figure from [Jain *et al* 87]:

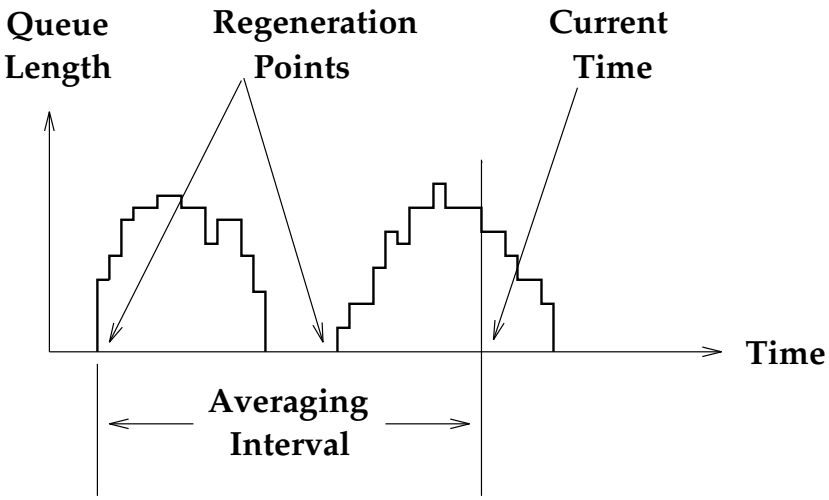


Figure 3: Interval for Queue Length Averaging [Jain *et al* 87]

As well as detecting congestion, a router must determine which set of sources should be asked to adjust their load, by setting the Congestion Experienced Bit in their packets. Sources taking more than their fair share of the resource have their bit set; other sources do not have their bit set.

A source reacts to the Congestion Experienced Bit by either increasing its window size by a constant (if the bit is not set), or decreasing the window size by a multiplicative factor. Once the window size changes, new Congestion Experienced Bits are ignored for one round-trip's time, to allow the traffic's effect on the network to be monitored and returned to the source via another Congestion Experienced Bit.

Jain et. al show that Dec Bit is robust and brings the network to a stable operating point with minimal oscillations in sources' window size. Since this paper, DecBit has been found to be a good congestion avoidance scheme. However, the convergence time can be several round-trip times, which does not help to alleviate short-term network congestion.

2.4.3 The Q-Bit Mechanism

An extension to DecBit, Q-Bit [Rose 92], suggests feeding a further bit of congestion information back to the transmitting source. This Q-bit is set when a packet received by a router could not be retransmitted immediately, but had to be queued for transmission. Rose states that this gives additional information to the source:

- when an overload-condition is approaching, a queue is building up and thus the Q-bit is set. This happens some time before C-bits [Congestion Experienced bits] will be set. Thus in this situation the sender should refrain from increasing its rate, keeping it constant instead.
- after the queue has vanished, C-bits may still last for a certain while. The Q-bit on the other hand will be turned off as soon as the queue disappears, and the sender should refrain from decreasing the load in that situation, keeping it constant again.

Sources react to either set C-bits or set Q-bits by lowering their rate. If the C-bit is set but the Q-bit is off, the source keeps its rate constant. The scheme also introduces a 'Freeze Time': a period after a source's window is reduced where no further windows reductions will occur. This helps to make the Q-bit scheme more stable when there are several concurrently transmitting sources.

In comparison with DecBit, Q-bit gives equal if not better results, and can guarantee fair division of bandwidth for traffic flows with different path lengths. The scheme is also useful for rate-based transport protocols as well as window-based protocols. As with DecBit, convergence to equilibrium still takes several round-trips to achieve.

2.4.4 Summary

Network-based schemes which feedback congestion information to sources give them direct information about the congestion state of the network. This should allow the sources to alter their transmission behaviour so as to reduce the congestion. The schemes described above have two main deficiencies. Firstly, only the existence (or the lack) of congestion is reported to the source: it cannot determine the extent or the severity of the congestion. Secondly, the sources use Sliding Window flow control, and it may take several round-trips for a source to reduce its window to a point which minimises congestion.

2.5 Packet Discarding Mechanisms

The main aim of congestion control is to reduce the rate of source transmissions to that which can be sustained by the network, thus preventing loss of data due to congestion. However, in the short term, routers must drop packets when buffers overflow. This is a form of congestion recovery.

Because Transport protocols like TCP increase their sliding window until packets are lost, there is usually at least one router along the path of a traffic flow which is congested, and has several packets queued for transmission. In this situation, a router may decide to drop packets *before* buffers overflow: this is a form of congestion avoidance. The choice of which packet to

drop can also reduce any unfairness in network usage (such as was asserted against TCP Vegas in Section 2.3.4).

A router typically chooses the most recently received packet to discard when buffers are full, and this is known as Drop Tail, as packets are normally queued on the tail of the router's buffer. This is not the only choice available to a router.

Nagle [Nagle 84] suggests a strategy of discarding the latest packet from the source that has the most packets buffered in the router. This would appear to penalise traffic flows overusing available bandwidth. In fact, a traffic flow may be underutilising the bandwidth between its source and destination, and still have the most packets buffered. Therefore, this scheme may penalise the 'wrong' traffic flows.

Another strategy would be to discard an incoming packet if it duplicates a packet which is already buffered in the router. This would provide some protection against poor Transport Layer implementations, or protocols where round-trip time estimation is poor.

2.5.1 Random Drop

Random Drop [Hashem 90] is a mechanism by which a router is able to randomly drop a certain fraction of its input traffic when a certain condition is true. The premise of Random Drop is that the probability of a randomly chosen packet belonging to a particular connection is proportional to the connection's rate of traffic. The main appeal of Random Drop is that it is stateless. A router using Random Drop does not have to perform any further parsing of packet contents to implement the mechanism.

Random Drop can be used as either a congestion recovery or a congestion avoidance mechanism. In the former, a router randomly drops packets when it becomes overloaded. In the latter, packets are randomly dropped to keep the router at its optimum power position.

Random Drop depends heavily on packet sources interpreting packet loss as an indicator of network congestion. Protocols such as TCP do make this interpretation. However, other protocols such as UDP do not make this interpretation. Therefore, Random Drop may not be useful in a heterogeneous source environment.

Another problem is that Random Drop does not distinguish between "well-behaved" and "ill-behaved" sources. Packets are dropped for sources that are congesting a router, and for sources that are not congesting a router. Similarly, for low-rate connections such as keyboard 'Telnet' connections, the loss of one packet may be a significant loss of the connection's overall traffic.

When Random Drop is used for congestion recovery (RDCR), instead of dropping the packet that causes the input buffer to overflow, a randomly chosen packet from the buffer is dropped. Mankin [Mankin 90] analyses Random Drop Congestion Recovery theoretically and as implemented in a BSD 4.3 kernel. She notes that RDCR is valuable only if the full buffer contains more packets for high-rate traffic flows than for low-rate traffic flows, and that packet arrivals for each source are uniformly distributed. In reality, Mankin notes that correlations such as packet trains and TCP window operations make the distribution non-uniform.

Floyd et. al [Floyd & Jacobson 91] note that Drop Tail can unfairly discriminate against some traffic flows where there are phase differences between competing flows that have periodic characteristics. Their analysis shows that Random Drop can alleviate some of this discrimination, and led to the development of Random Early Detection.

2.5.2 Random Early Detection

Random Early Detection (RED) [Floyd & Jacobson 93] is a form of Random Drop used as congestion avoidance. A RED router randomly drops incoming packets when it believes that is is becoming congested, implicitly notifying the source of network congestion by the packet loss. RED is essentially designed to work with TCP.

Floyd and Jacobson's results indicate that Random Early Drop works well as a congestion avoidance scheme, providing fair bandwidth allocation and coping with bursty traffic better than DecBit. They also note that Random Early Drop is reasonably simple, appropriate for a wide range of environments, and does not require modifications to existing TCP implementations.

It is important to note, however, that Random Early Drop does not work with non-TCP protocols, still causes packet loss, and still takes time to find equilibrium. For transport protocols which can interpret the DecBit C-bit, Floyd and Jacobson note that RED can be used as a DecBit replacement where the C-bit is set instead of dropping the packet.

2.5.3 Summary

Loss of packets (intentional or otherwise) is considered detrimental, as work has been wasted to get the packet where it is within the network. However, if packet loss is unavoidable, packet discarding mechanisms can help to even out any unfairness in network resource usage. Where Transport protocols use packet loss to infer congestion, intentional packet dropping can be used as a form of congestion avoidance.

2.6 Packet Queueing Mechanisms

If a packet received by a router cannot be immediately retransmitted, it must be queued for retransmission. If a router is congested and has a number of packets already queued for transmission, the position where the new packet is inserted in the queue can have a positive congestion control effect, and can reduce unfair resource utilisation. In effect, a packet queueing mechanism can also be considered a queue reordering mechanism. Several queueing mechanisms have been studied which have an effect upon network congestion.

The standard packet queueing mechanism is First-In-First-Out, or First-Come-First-Served. Packets are removed from the retransmit queue (and retransmitted) in the same order that they are queued. FIFO is flow-neutral; it does not distinguish between flows with different characteristics.

2.6.1 Nagle's Fair Switching

In [Nagle 87], Nagle suggests an alternative to FIFO queueing. Take a router with two or more network interfaces. Congestion can occur if packets arrive from one interface faster than they can be dispatched to the other interfaces. Nagle replaces the packet queueing mechanism with a round-robin approach, choosing packets from queues based on the source address of each traffic flow. If a queue is empty, it is skipped. Thus, each flow gets equal access to the output interface; if one source transmits packets faster than the router can process them, it does not affect the

available bandwidth for other sources: instead, the queue for the source increases, affecting its own performance.

Nagle concludes that the proposed method of packet scheduling only makes access to packet switches fair, but not access to the network as a whole.

2.6.2 Fair Queueing

A problem with Nagle's Fair Switching is that the variation in packet sizes are not taken into account. A traffic flow with 1500-octet packets gets three times as much bandwidth as a flow with 500-octet packets. Fair Queueing [Demers *et al* 89] works like Nagle's scheme in that packets are separated into separate per-flow queues, and are serviced in a round-robin fashion, but packet size variation is taken into account.

Fair Queueing simulates a strict bit-by-bit round robin queueing scheme by determining at which time each packet would finish being transmitted if done in a bit-by-bit fashion. This time is used to schedule the packet transmissions from each of the per-flow queues. Once a packet's transmission commences, it cannot be interrupted. Therefore, Fair Queueing only approximates a strict bit-by-bit round robin queueing scheme.

The effect of Fair Queueing is to limit each of N traffic flows to $1/N$ of the available output bandwidth, while there are packets in each of the per-flow queues. This prevents greedy traffic flows from obtaining an unfair amount of output bandwidth. Of course, when some per-flows are empty, they are skipped, so that the $1/N$ limit can be exceeded.

Other packet queueing mechanisms, such as Hierarchical Round Robin [Kalmanek *et al* 90], Fair Share [Shenker 94], FQNP and FQFQ [Davin & Heybey 90], have been proposed which also address the issue of fairness in resource usage.

2.6.3 Summary

Packet queueing schemes can help to ensure network bandwidth is distributed fairly, but only when buffer occupancy within routers is high, past their optimum point of operation. These schemes affect the choice of which packets to drop, and thus implicitly feed back congestion information to badly-behaved traffic sources. Packet queueing schemes, thus, are not congestion control schemes themselves, but can help alleviate congestion in conjunction with existing control schemes.

2.7 Summary

Several classes of congestion control and fair resource allocation schemes have been reviewed in this chapter. Nearly all of the control schemes have been congestion recovery schemes. Traditional Transport protocols such as TCP probe for excess bandwidth by increasing transmission rates until the network becomes congested. Only when implicit congestion information (such as packet loss) is perceived is the transmission rate reduced.

Network-based schemes which monitor the congestion state of the network and return this information to the source have been proposed and implemented. Because the Transport Layer

now receives explicit congestion information, it can react to congestion more quickly. However, several round-trips are often required to reach equilibrium.

Within the network, routers with high buffer occupancies can choose different packet dropping or queueing schemes to reduce congestion and distribute resources more fairly. The drawback here is that packets are typically discarded, even with congestion avoidance schemes. This lowers the throughput of sources and wastes the resources used to forward the packet to the point where it is discarded.

The traditional control schemes reviewed suffer several drawbacks. End-to-end schemes may not utilise network resources fairly: traffic flows compete for resources, and the network does not enforce fairness. This can be partially addressed with packet queueing and packet dropping schemes. Sliding window flow mechanisms can admit bursts of packets into the network which cause short-term congestion. Finally, congestion can be detected only by end-to-end packet loss, increases in delays, or via congestion indicators such as DecBit and Source Quench: two of these indicators can be inaccurate, and the time to react to these indicators can be several round-trip times. [Eldridge 92] notes that "window-based ... congestion controls could be effective under long network delays but [are] ineffective under short network delays". The next chapter will review some rate-based control schemes which attempt to address the deficiencies outlined in this chapter.

Chapter 3

Rate-Based Congestion Control Schemes

3.1 Introduction

We are considering best-effort connectionless packet-switched networks where link capacity is typically fixed. Given that link bandwidth (and hence overall bit rate) is fixed, network operations which deal with bit rates may be useful. For example, if routers can feed back rate information to sources of traffic flows, then the routers can participate in the fair allocation of link capacity. Transport protocols which are rate-based can admit packets into the network uniformly spaced: this helps to prevent short-term congestion. Combined, these techniques can be used as a form of congestion control, by allocating rates to traffic flows which keep network operation at the knee-point of peak power (see Figure 2, pg. 4).

There is less research into the use of rate-based techniques than for non-rated-based techniques in connectionless packet-switched networks. This chapter examines some of this research, and also reviews the use of rate-based techniques in other network architectures such as Asynchronous Transfer Mode (ATM).

3.2 Rate-Based Flow Control

In the late 1980s, several new transport protocols for connectionless networks were proposed: VMTP, NETBLT and XTP. These protocols featured end-to-end rate-based flow control instead of the more traditional sliding window flow control.

3.2.1 NETBLT

In [Clark *et al* 87], Clark et al describe the NETBLT transport protocol. Their previous experience with transport protocol performance indicated two major problems: "a restriction in throughput which arises from the use of windows as a flow control mechanism, and the difficulty in handling timers". They note that "[data] transmission ought to be evenly distributed over a [round-trip time] period to match the receiver's consumption speed. Unfortunately, windows only conveys the latter – *how much* data can be buffered, rather than the latter – *how fast* the transmission should go." The difficulty with timers is, as was noted in Section 2.3.3, due to the high variance in round-trip time. They concluded that:

- Windows and timers perform poorly in synchronizing the end state. To achieve

fast resynchronization of the end state, we need better mechanisms than the cumulative acknowledgment, and we must reduce reliance on timers.

- Flow control must be independent from error control. Mixing the two in one mechanism can only make the flow control vulnerable to transmission errors and delays.

The NETBLT transport protocol replaces the traditional window flow control with a rate-based flow control mechanism or, more specifically, a coarse-grained rate-based mechanism: packets are transmitted in *groups*. NETBLT's rate control has two components, a *burst size* in packets which sets the group size, and a *burst interval*. This interval is the period between the transmission start of successive bursts. Selective acknowledgments are used for error control.

At connection setup, an initial burst size and interval is negotiated between source and destination. These are renegotiated by the destination in the selective acknowledgment of the received group[1]. Experimental results over a low round-trip 10Mbps LAN and a high round-trip (1.8 seconds) 1Mbps WAN showed that such a coarse-grained rate-based flow control mechanism showed promise. Clark et al noted, however, that a protocol using rated-based flow control "must be able to set an initial transmission rate based on the available network bandwidth and the speed at which the receiver can process data. ... It is therefore important that support for rate selection exist at the network level."

3.2.2 XTP

The Xpress Transfer Protocol [Sanders & Weaver 90] provides selective acknowledgment error control and group-oriented rate-based flow control in a fashion similar to NETBLT. The aim of XTP's development was to streamline the processing requirements of a transport protocol to the point where it could be "implemented in hardware as a VLSI chip set".

The protocol's headers set a *burst rate*, the maximum number of bytes the receiver will accept per second, and a *burst size*, the maximum number of bytes the receiver will accept per burst. As the aim of XTP is to reduce processing requirements, [Sanders & Weaver 90] gives no further details on its rate-based flow control.

3.2.3 VMTP

The Versatile Message Transport Protocol [Cheriton 86] [Cheriton & Williamson 89] was designed to overcome perceived deficiencies in the performance and functionality of traditional network protocols. Like NETBLT and XTP, VMTP provides selective acknowledgment error control and group-oriented rate-based flow control. VMTP's flow control is more fine-grained than NETBLT or XTP. While packets are still transmitted in bursts, the overall burst rate is controlled by the burst size and the *inter-packet delay*. Without the inter-packet delay, "a client must be prepared to accept an entire packet group at once". Cheriton et al suggest that "if the receiver requests retransmission of every fourth packet, the sender can reasonably increase the inter-packet gap. Moreover, the sender can periodically attempt to reduce the inter-packet gap when no packet loss occurs, to ensure that it is transmitting at the maximum rate the receiver can handle." As with XTP, the immense functionality of VMTP prevents any further description of the rate-based flow mechanism.

[1] The mechanism describing this negotiation is not given.

3.2.4 Summary

These three studies demonstrate that protocols can be built for connectionless networks where error and flow controls are separated, and that rate-based flow control is feasible. As with all end-to-end protocols, these protocols do not attempt to measure or control congestion *within* the network, but only limit the source's transmission characteristics to that which can be sustained by the destination.

3.3 Available Bit Rate Congestion Control in ATM Networks

Asynchronous Transfer Mode (ATM) networks are characterised as connection-oriented cell-switched networks [Partridge 94]. Cells are fixed in size (53 octets), and virtual circuits are established across the network to propagate the traffic for each connection. ATM provides several service classes to the connections established between sources and destinations: Constant Bit Rate (CBR), Variable Bit Rate (VBR), Available Bit Rate (ABR) and Unspecified Bit Rate (UBR). Of the four service classes, only ABR provides service degradation congestion control [Jain 96], which is the type of congestion control under consideration in this thesis.

With ABR, a source of cell traffic can specify minimum and maximum cell rates at connection establishment. If the connection is granted, then the source can transmit at a rate between the specified minimum and maximum cell rates. However, if the network becomes congested, the source may need to reduce its current cell rate to minimise congestion.

Several rate-based congestion control mechanisms were proposed to control source cell rates for the ABR service. The main proposals are summarised below.

3.3.1 BECN

Early congestion proposals were Forward Explicit Congestion Notification (FECN) [Newman 94] and Backward Explicit Congestion Notification (BECN) [Newman 94]. In BECN, the intermediate switches[2] return special *resource management* (RM) cells to sources of cells if they believe the source is causing congestion. Congestion is identified by monitoring cell queue lengths within the switches. On receipt of an RM cell, a source is obliged to halve its transmission rate.

If no RM cells arrive after a 'recovery period', the source may double its cell rate (without exceeding the maximum cell rate specified at connect time). The recovery period is proportional to the current cell rate: as the cell rate drops, the recovery period shortens.

BECN can be seen to be analogous to the Source Quench scheme, controlling the flow of a specific ATM connection.

3.3.2 PRCA

Another early proposal for ABR congestion control was a modification of DecBit. The destination returns RM cells to the source if, over a certain period, any Congestion Experienced Bits are set.

[2] ATM switches perform cell routing, and so are analogous to routers in packet-switched networks.

The source uses Multiplicative Decrease and Additive Increase to control its cell rate.

One problem noted with this approach [Jain 96] is that, under heavy congestion conditions, the RM cells which cause rate decrease may not reach the source, and so the congestion may not be reduced. A modification to the proposal, known as the Proportional Rate Control Algorithm (PRCA) [Hluchyj 94], has the source set every Congestion Experienced Bit on, but leave one in N off. If any RM cells arrive with the Congestion Experienced Bit off, then the destination sends RM cells back to the source, which prompt a source rate *increase*. Again, Multiplicative Decrease and Additive Increase is used to control the source's cell rate. With this approach, the source cannot increase its rate unless RM cells are received.

PRCA was found to have a fairness problem for long-path connections. If the probability of a cell having its Congestion Experienced Bit set by one ATM switch is p, then the probability of it being set over a p-switch path is p^n. Thus, long-path connections have a lower chance of receiving a rate increase RM cell than short-path connections. This is known as the 'beat-down problem' [Bennett & Des Jardins 94].

3.3.3 EPRCA

One problem with binary schemes like BECN and PRCA is that the time to reach an optimum cell rate is long: several iterations of Multiplicative Decrease and/or Additive Increase may be required to bring the source to the optimum cell rate. [Charney *et al* 94] suggested returning an explicit cell rate value to the sources of cells. This rate is set using the 'Fair Share' algorithm described in [Charney 94], and is covered in detail in Section 3.4.

The use of explicit cell rate information has several advantages: switches can monitor for violations of the cell rates, and convergence to the optimum cell rate is shorter than for the binary schemes. The loss of cells carrying explicit cell rate information has less impact than the loss of BECN or PRCA RM cells: the source should receive other cells carrying explicit cell rate information to set its optimum cell rate.

Enhanced PRCA (EPRCA) [Roberts 94] resulted from a combination of PRCA and the explicit cell rate scheme described above. Cell sources send a combination of RM cells and data cells with Congestion Experienced Bits. Switches along the path of a connection may calculate the Fair Share rate for the connection (and place the value in the RM cells), set the Congestion Experienced Bits if the connection is exceeding its fair share (or is causing congestion), or both.

EPRCA allows switches to perform binary-feedback congestion control, explicit rate congestion control, or both.

3.3.4 Summary

Although ATM is a very different network architecture to the connectionless packet-switched networks under consideration, many congestion control techniques are applicable to both. Analogues of both Source Quench and DecBit have been proposed for the ATM ABR service: as well, schemes which provide explicit rate information to sources have been proposed. It may be possible to apply the latter techniques to packet-switched networks. However, this would require such modifications as rate-based Transport protocols, operation with variable-sized packets and the identification of source/destination flows by the Network Layer.

3.4 Charney's Thesis

The EPRCA congestion control scheme depends upon the 'Fair Share' rate allocation algorithm. [Charney 94] describes this algorithm, which is very similar to the RBCC algorithm described in Chapter 6.

The 'Fair Share' algorithm is performed by the routers in a rate-based framework where some packets in a traffic flow contain rate fields, which are adjusted by the algorithm, and then fed back to the traffic source to control its rate. The algorithm effects the requirements of the *Maximin* optimality criterion, described informally in [Charney 94]:

> Consider a network with given link capacities, the set of sessions and fixed session routes. We are interested in such rate allocations that are feasible in the sense that the total throughput of all sessions crossing any link does not exceed the link's capacity. We would like the feasible rate allocation to be fair to all sessions. On the other hand we want the network to be utilized as much as possible.
>
> We now define a fair allocation in the following way. We consider all "bottleneck" links, i.e the link with the smallest capacity available per session. [...] We share the capacity of these links equally between all sessions crossing them. Then we remove these sessions from the network and reduce all link capacities by the bandwidth consumed by the removed sessions. We now identify the "next level" bottleneck links of the reduced network and repeat the procedure. We thus continue until all sessions are assigned their rates.
>
> Such a rate vector is known as *maxmin fair* allocation.

The Fair Share algorithm is described in some detail in [Charney 94] (pages 24-32). The following description is taken from [Jain 96]:

> The fair share is computed using an iterative procedure as follows. Initially, the fair share is set at the link bandwidth divided by the number of active VC's [virtual circuits]. All VCs whose rates are less than the fair share are called "underloading VCs". If the number of underloading VCs increases at any iteration, the fair share is recomputed as follows:
>
> $$Fair\ Share = \frac{Link\ Bandwidth - \Sigma\ Bandwidth\ of\ Underloading\ VCs}{Number\ of\ VCs - Number\ of\ Underloading\ VCs}$$
>
> The iteration is then repeated until the number of underloading VCs and the fair share does not change.

Charney proves that, given an upper bound D on round-trip delay, and the number N of active virtual circuits, the upper bound on convergence time from any initial conditions is no more than $4ND$. Simulations in [Charney 94] show that the bound is more realistically $2ND$.

A comparative discussion on Fair Share and RBCC is delayed until Section 7.6.

3.5 Theoretical Work

One difficulty with congestion control schemes is to verify that they will control congestion over the set of all possible network designs, traffic sources and usage conditions. Some research has been undertaken to prove rate-based congestion control schemes within a limited set of conditions. This research shows that rate-based schemes can control congestion well, and within a short space of time.

[Benmohamed & Meerkov 93] analyse a rate-based control scheme which monitors the occupancy x of router packet buffers against a desired occupancy x^0. Routers periodically calculate an admission rate for sources based on the measured difference between x and x^0, and return the rate to traffic sources, who in turn are obliged to lower their rate to the lowest value obtained.

Analysis is given for a backbone router with one congested outgoing link. They derive formulae to determine a set of fair admission rates for all traffic sources through the congested node:

> If M connections share a transmission link, then each gets $1/M$ of the bandwidth, and if $(M - l)$ connections use less than their share, then the unused portion is equally distributed among the rest.

[Benmohamed & Meerkov 93] note that the interval T between measurements of x affects the performance of the scheme. If T is small, x converges more quickly to x^0 with less buffer overshoot, but also causes higher communication overhead within the network, and more work for the router. As well, the communication delays to the source of each traffic flow, and the round-trip times for each traffic flow, affect the choice of T. Experimental work shows that the congestion scheme does cause x to converge rapidly to x^0 with some oscillation, which may be further reduced by using adaptive techniques.

A similar scheme is proposed by [Fendick *et al* 92], where both the level of occupancy and the arrival rate of packets is measured, and with a desired operating point of $x^0 = 1$. They show that their scheme rapidly converges to the desired operating point, although they noted that the scheme is locally unstable in the region of this operating point. The scheme deals with the effect of both delay due to packet propagation and randomness in the feedback delay.

An alternative rate-based control framework which is end-to-end based is described in [Haas 91]. Here, the source measures the interval between pairs of transmitted packets. The destination also measures this interval as packets arrive. This interval, termed the "virtual delay", is the difference between the interval as measured by the source and destination. The "virtual delay" is averaged over several packets and is returned, along with measured packet arrival rates, to the source. The source uses this data to estimate a level of congestion, β, such that $\beta = 1$ indicates low congestion and $\beta = 0$ indicates high congestion. A step function is used to convert changes in "virtual delay" into values of β.

Sources admit traffic using a scheme like Leaky Bucket [Turner 86] which is controlled by the estimated congestion level. The inter-packet gap is calculated using

$$interpacket\ gap = min(\frac{1}{\beta}, quota) - 1$$

where $quota$ is the interval, in packet transmission time units, between updates of β. When $\beta = 1$, the inter-packet gap is zero, and when $\beta = 0$, the inter-packet gap is $quota$.

The framework allows for many different methods to determine β and the update interval $quota$. Haas notes that the value of $quota$ depends on the network topology and the network traffic, but that the number of "virtual delay" samples per $quota$ should be large enough to minimise statistical noise. A future approach would be to use adaptive techniques to match the values of all the parameters to the network topology and traffic.

Yet another rate-based congestion control alternative, Hop-by-Hop, is described in [Mishra *et al* 96], where service rates of traffic flows are adjusted by routers based on information provided by neighboring routers. Each router monitors its buffer occupancy for all active flows, and periodically informs upstream routers of this information. Upstream routers use this information to compute a *target* sending rate for each flow. Rather than aiming for the ideal of one packet queued per router, Hop-by-Hop keeps a small number of packets queued at each router, which can be transmitted if additional bandwidth becomes available downstream.

Hop-by-hop is analysed over a set of N routers which form a single route: all flows take this single route. Cross-traffic to the flows along the route is also modeled. Analytical results show that the scheme brings flow rates and desired queue occupancies to the desired operating point with no instabilities.

3.5.1 Summary

This theoretical research into rate-based congestion control schemes shows that congestion control can be successfully implemented using rate-based techniques, with fast convergence of network conditions to near the desired operating point. The schemes presented, however, describe simple models or experiments where there is only one congested node, and/or the paths for all traffic sources take the same route. The routers must periodically sample the traffic characteristics of the traffic flows, and the interval between periods affects the performance of the congestion control.

3.6 Non-Service Degradation Control Schemes

Most of the congestion control schemes reviewed so far are service reduction schemes: sources must reduce their transmission rates to reduce congestion. An alternative to this is to identify the existence and extent of network congestion; it is then up to each source to decide what the most suitable response is to the identified congestion conditions.

Williamson & Cheriton [Williamson & Cheriton 91] propose a congestion control schemes which returns a *Loss/Load* curve to the source. This curve gives the probability of packet loss as a function of the source's offered load (as a rate) on the network. This has several advantages over service degradation control schemes:

- The network can provide a predictable level of service without requiring resource reservations, allowing data to be transmitted without connection setup delays.
- The source can use the Loss/Load curve to choose an operating point which is acceptable in terms of throughput and packet loss.

The Loss/Load curve is monotonically increasing: as offered load increases, so will the probability of packet loss. The network is free to provide different Loss/Load curves to different sources, based on an operating policy. The curve can be treated as a 'contract' between the network and the source for a specific time interval.

When the load offered to a router is greater than its available output bandwidth, the router must discard some of the load. There are many different ways to do this; the choice of method is part of the operating policy and affects the Loss/Load curves returned to the sources. The queueing scheme is arbitrary, and the router may use hints on traffic type to bias packet loss (e.g to favour loss-insensitive traffic). The router monitors the rates of traffic flows over an interval using an exponentially weighted average. The Loss/Load curves are calculated and transmitted periodically, or when the load on the router changes significantly. They are not transmitted when packets are lost.

The host receives a number of curves from the routers along the traffic path. It uses the Loss/Load curve for the most congested router, and uses this curve to set its transmission rate. Leaky Bucket [Turner 86] or Virtual Clock [Zhang 91] can then be used for packet admission.

The aim of the Loss/Load congestion scheme is not to eliminate packet loss, but to allow each source to control the amount of loss during transmission. One consequence of the scheme is that, as loss increases as offered load increases, a source's throughput reaches a maximum at a load value before packet loss reaches a probability of 1. Therefore sources which are greedy and transmit above this load value will decrease their overall throughput. Simulation results in [Williamson & Cheriton 91] show that the scheme does give accurate packet loss probabilities, and the Loss/Load curves converge to equilibrium after two to four iterations.

3.7 Summary

Rate-based congestion control schemes form an alternative option to the traditional end-to-end and window-based schemes reviewed in the previous chapter. They are particularly useful in such networks as ATM where connection establishment and admission of data into the network are rate-based.

Rate-based control schemes allow intermediate network nodes to participate in the control of congestion, by monitoring congestion and returning information on congestion to traffic sources. This feedback may be a single bit (BECN), an explicit rate value (PRCA and EPRCA), or even a probability of packet loss (Loss/Load).

Theoretical studies of rate-based congestion control schemes show that they can bring source transmissions to the optimal point of operation within a small number of iterations. In general, these studies have been limited to simplistic network architectures, and may not model global network behaviour.

The next chapter will examine a rate-based framework for congestion control in connectionless packet-switched networks. The framework attempts to address the deficiencies in traditional control schemes via the use of explicit rate feedback to sources of network traffic.

Chapter 4

A Rate-Based Framework for Congestion Control

The previous chapters show that, despite the advances made in terms of congestion control, traditional protocols and router mechanisms are deficient in terms of minimising congestion. Rate-based congestion control techniques may be useful in overcoming some of these problems. In this chapter, I give a brief discussion of the deficiencies in traditional congestion control schemes, and describe possible remedies. Given the remedies proposed, I sketch a framework for congestion control which is constructed on a 'sustainable rate' basis, without use of such paradigms as sliding windows and round-trip timers. I then examine the components of the framework in detail, and look at the parameters inherent in the framework.

4.1 Deficiencies and Remedies

4.1.1 End-to-End Control Mechanisms

End-to-end congestion control mechanisms do not perform well because they must infer network congestion from indirect evidence such as packet loss and round-trip delay increases. In many cases, these inferences are wrong: packets are lost due to other reasons such as packet corruption. Mechanisms which feed congestion information from routers back to traffic sources seem to work more effectively than purely end-to-end mechanisms.

The 'sliding window' mechanism is often overused. In protocols such as TCP, sliding windows performs error recovery, end-to-end flow control, and a primitive congestion control. The solution to these three problems is thus not orthogonal. Fast changes in window size can lead to bursty source transmissions: This can cause both short-term and long-term congestion in routers and destinations.

Most end-to-end mechanisms require a round-trip timer to detect packet loss: this is a problem in itself. Due to the often high variance in round trip time, the timer is usually set too low or too high, causing unneeded packet retransmission or poor throughput, respectively. Its inaccuracy leads to incorrect end-to-end judgements of current levels of congestion, based on packet loss. A round-trip timer is also computationally expensive to implement.

To avoid these shortcomings, transport protocols should ensure that error recovery, flow control and congestion control are completely orthogonal. Sliding windows certainly should not be used for congestion control. True rate control should be used for both flow control and conges-

tion control, and packet admission into the network should be as evenly-spaced as possible. This minimises the burstiness of traffic, and helps to minimise short-term router congestion.

A protocol using true rate control should obtain information on the preferred rate from any congestion control scheme in the underlying Network Layer. Error detection should be performed by Selective Acknowledgments and Selective Retransmission. This retransmits only those packets that have been lost and need retransmission.

Instead of a sliding window to limit in-transit traffic for error recovery, a transport protocol should be allowed to have as much in-transit traffic as it desires. This is limited only by its transmission rate, available memory, and the amount of application data ready to transmit. A round-trip timer is still required, but only when a source exhausts its supply of traffic to transmit and must wait for acknowledgments. Until the protocol runs out of data, it should be transmitting in a rate-controlled regime that ensures good throughput. As the timer's value is not critical, its estimation will not be computationally expensive. Furthermore, it will be rarely used, and will thus cause little overhead.

4.1.2 Congestion Detection Schemes

The ultimate aim of any congestion scheme is to reduce the load on the network to that which is sustainable. Simplistic schemes such as Source Quench only inform the source of the existence of congestion within the network: no other information can be deduced.

Protocols such as Dec Bit are an improvement over Source Quench in terms of their detection of congestion and the return of this information to sources. However, again, only the existence of congestion is returned to the sources.

Schemes such as RED, while admirable in attempting to address the problems of end-to-end congestion control, can only indicate congestion by *dropping* packets. Packets should not be discarded unless absolutely necessary.

If transport protocols choose to employ rate control as a congestion control method, a congestion scheme should attempt to determine the ideal rate of flow for each source. If these ideal rates can be returned to all sources, the adjustment of their rates should alleviate any congestion.

4.1.3 Router Procedures

The aim of any router is to forward traffic and maximise power, operating at the knee of the power diagram in Figure 2 (pg. 4). Power is maximised when response time is low and throughput is high.

Low response time occurs when network delays are low, such as when router buffer occupancy is low. Throughput is high when packets are not lost and links are fully utilised. Power is maximised when an incoming packet can be immediately retransmitted without being delayed due to buffering. Router buffer occupancies should therefore never exceed one packet.

Congestion mechanisms which depend on high buffer occupancies, such as packet reordering schemes and packet dropping schemes, can only be useful in exceptional circumstances when congestion recovery is required. A good congestion control scheme will keep the network operating at maximum power, with low buffer occupancies.

4.1.4 Packet Admission Mechanisms

Bursty packet admission (from a traffic source or from an intermediate router) will make low router buffer occupancies difficult to achieve, and will also cause short-term packet delays due to buffering. Smooth packet admission techniques such as Leaky Bucket [Turner 86] should be used by traffic sources and routers to help reduce short-term congestion [Sidi *et al* 93]. If congestion recovery is required, router packet reordering schemes which smooth traffic admission may also be employed.

4.2 A Rate-Based Framework: Design Considerations

The essential thrust of the remedies just given is to obtain detailed congestion information from the network, and to use this to adjust transmission rates which will optimise overall network power. The obvious paradigm for such a congestion control system is to be rate-based.

During the development of my rate-based congestion control framework, I was guided by the following design decisions. These address the deficiencies that I have outlined in existing congestion control schemes.

Transport congestion control should use congestion information from routers. Bit-setting techniques help a source identify the presence of congestion, but do not indicate the extent of congestion. Routers should feed back more than one bit of congestion information to transport protocols. Ideally, the routers should provide transport protocols with a *sustainable* rate for each traffic flow which allows optimum throughput without causing network congestion.

Dropping packets is highly undesirable. A router which is dropping packets is operating in congestion recovery mode, well past the optimum point of operation [Jain 90]. A congestion control scheme should emphasise congestion avoidance, but also provide congestion recovery. If routers are providing sources with congestion information, it is even more important that packets should not be dropped if at all possible.

Router queue sizes $>$1 are highly undesirable. Jain [Jain 90] shows that an average router queue length of 1 packet is the optimum value. Queue lengths higher than 1 cause increasing end-to-end delays and round-trip times. Variable queue lengths also add variance to these times. Increasing queue lengths indicate that the router is congested: more packets are arriving than can be retransmitted. A congestion control scheme should strive to keep average queue lengths at length 1.

Use of round-trip timers is undesirable. As we have seen, round-trip time estimation is complex and difficult to get right. The proposed congestion control framework should avoid the use of round-trip timers if at all possible.

Use of sliding windows for flow control is undesirable. Window update schemes are complex and difficult to get right. Window updates can also lead to bursty traffic, undesirable both from the point of view of congestion and time-sensitive network traffic. Limited window sizes can also lead to poor utilisation in high-speed networks with large latencies. Therefore, window-based flow control techniques should be avoided.

Packet retransmission should not be done unnecessarily. Traditional packet retransmission methods such as Go-Back-N retransmit data that have been successfully received by the destination. Newer retransmission schemes such as Selective Retransmission only retransmit

data if it is known to have been lost. It is preferable to retransmit data only if required, as this helps to reduce the overall load on the network.

Separation of error control and flow control is desirable. Packet loss should not be associated with congestion or flow control in a transport protocol at all. Packets are lost for many reasons other than congestion. A transport protocol should use packet loss to prompt packet retransmission and nothing else.

Packet admission and readmission should be smooth. Bursty traffic (such as is seen during window updates) not only causes short-term congestion but increases end-to-end and round-trip variance. Transport protocols should admit packets into the network as evenly spaced as possible, and routers should attempt to preserve this spacing. Techniques such as Leaky Bucket should be used here, and any packet reordering should be discouraged. In fact, queue reordering is essentially impossible if router queue sizes are kept around one packet.

These design decisions form the requirements for the components in the congestion framework.

4.3 Overview of the Framework

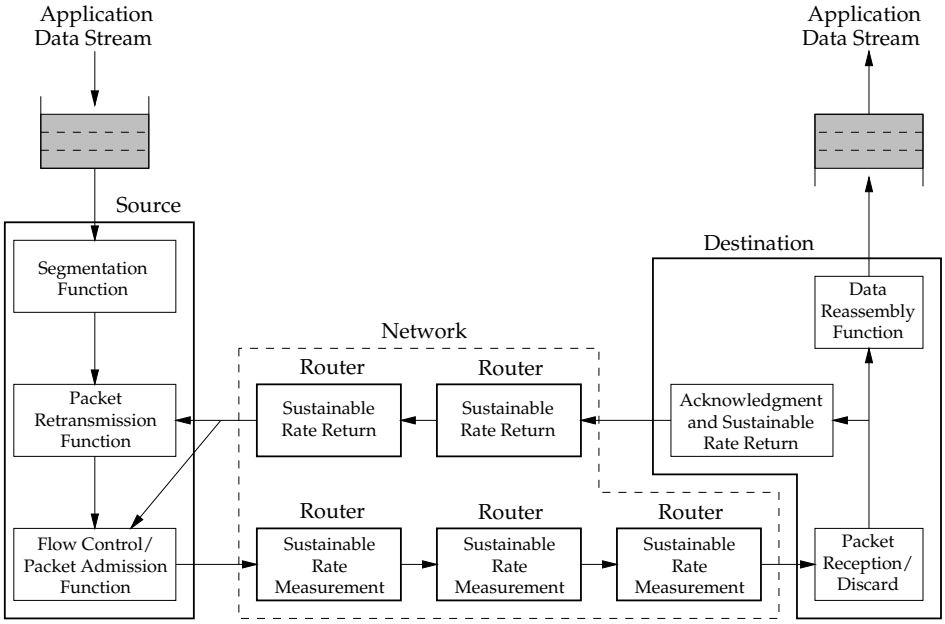


Figure 4: Overview of the Rate-Based Congestion Control Framework

The framework that I propose takes into account all of the requirements just discussed. In essence, the framework spreads the mechanisms of congestion control across both the Transport and Network Layers, measuring and using information on *sustainable rates of traffic flow* through the network. Figure 4 gives an overview of the framework, showing the main components. Routers consist of the functions shown in Figure 1 (pg. 3) but with an extra function, the Sustainable Rate Measurement function.

The essential elements of my proposed rate-based congestion control framework are as follows:

1. The Transport Layer uses a rate-based flow control scheme, using rate information provided by the Network Layer. The flow control and error control mechanisms are orthogonal, being performed by the Flow Control and Packet Retransmission functions, respectively.
2. A source admits packets for each traffic flow (i.e a single data stream from an application) into the network uniformly spaced in time. This minimises short-term congestion due to bursty traffic: this is performed by the Packet Admission function. Routers should attempt to preserve the time spacing of packets in a traffic flow.
3. The routers in the Network Layer implement a congestion control scheme, part of which measures the sustainable traffic rate across the network for each traffic flow; this is performed by the Sustainable Rate Measurement function. This sustainable rate measurement is passed to the destination machine and then via acknowledgment packets to the source machine. The idea of measuring the sustainable rate is the core idea to the framework, and the specific algorithm that I propose is named RBCC[1]. It is described in detail in Chapter 6.
4. Routers implement both congestion avoidance and congestion recovery mechanisms as the rest of the Network Layer congestion control scheme. Routers partition the available resources fairly amongst traffic flows, in both uncongested and congested operating regimes. Routers drop packets when in the operating region of the congestion cliff. These operations are performed by the Packet Selection, Packet Queueing and Packet Dropping functions.

4.4 Network Layer Rate Measurement

Before discussing the individual components in the framework, we need to look at the data flows that allow the components to intercommunicate. In the framework, the network routers must measure the sustainable traffic rate across the network for each traffic flow, and return the measurement to the source. This is best achieved with a number of fields in the Network headers of each packet sent from and to the source.

In the direction of flow from source to destination (e.g in data packets), I propose that three fields be used:

Desired_Rate: The bit rate which the source of this traffic desires: it can be ∞.

Rate: The bit rate which has been allocated to this flow of traffic by a router.

Bottle_Node: The router which set the value of the **Rate** field in the packet.

When a source transmits a packet, it initialises the **Desired_Rate** field to the bit rate which it desires for the flow of traffic; The **Rate** field is also initialised to this value, and the **Bottle_Node** field is set to the identity of the source. A finite **Desired_Rate** indicates that the source knows in advance the maximum data rate required for its transmission. A **Desired_Rate** of ∞ indicates that the source will use as much bandwidth as is made available to it.

As the packet passes through a router, the Sustainable Rate Measurement function may lower the value of the **Rate** field if it has allocated a lower rate to the traffic flow than the current value

[1] Rate-Based Congestion Control.

of the **Rate** field. If it alters the **Rate** field, the **Bottle_Node** field is set to identify the router. When the packet reaches its destination, the **Rate** field holds the lowest sustainable rate allocated by the routers along the packet's path, and the **Bottle_Node** field identifies the node that set this lowest rate.

In the direction of flow from destination to source (e.g in acknowledgment packets), I propose that one field be used:

Return_Rate: The value of the last **Rate** field which reached the destination.

This field is not altered as the packet travels back to the source. Thus, when the source receives the packet, it obtains a measure of the lowest bit rate available to it for the flow of traffic. It can then adjust its rate to not exceed this value, so as to minimise the congestion that the traffic flow causes to the network.

Note that acknowledgment packets carry not only the **Return_Rate** field but also the three fields **Desired_Rate**, **Rate** and **Bottle_Node**. This is because data packets and acknowledgment packets form *two* traffic flows, and they require different amounts of bandwidth and in different directions, as shown in Figure 5. Thus data packets carry the 'return' field to supply the destination with details of its acknowledgment traffic flow.

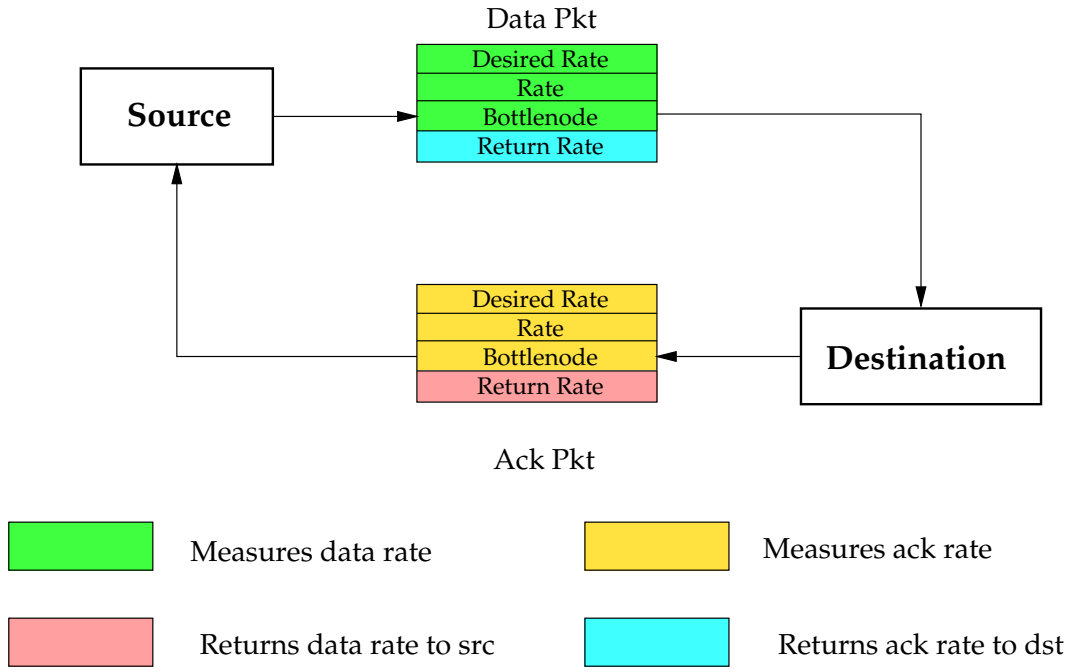


Figure 5: A Bi-directional Traffic Flow

4.5 Source Flow Control and Packet Admission Functions

In order to best utilise the **Return_Rate** values from the network components, and in order to admit packets as evenly spaced as possible into the network, the source should use a packet admission scheme. I propose a variant of Leaky Bucket [Turner 86]. The bucket has infinite buffer capacity, and admits packets into the network so that, at packet admission time, the flow's

bit rate is exactly the **Return_Rate**. The inter-packet delay, shown in Figure 6, is calculated using the equation

$$delay = \frac{packet\ size\ in\ bits}{Return\ Rate\ in\ bits\ per\ second} \tag{4.1}$$

Note that if packets are not constant in size, the delay will vary in value, even when the **Return_Rate** value is constant. The delay between packets can be greater than the value determined by Equation 4.1 if there is no available data from the application to transmit. However, once data becomes available, the source must not transmit packets faster than the set rate in order to "make up the average".

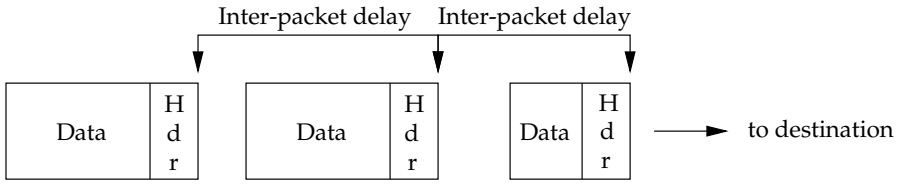


Figure 6: Inter-packet Delay in the Framework

Return_Rate values arrive in every acknowledgment packet from the destination. Immediate use of every value is liable to cause short-term rate fluctuations which could result in short-term congestion. To avoid this, the source must leave 1 round-trip gaps between delay changes, in order to allow the **Return_Rate** value used to take effect. Note that sequence numbers can be used to determine that a round-trip has occurred: a round-trip timer or round-trip time estimation is not required.

Although packets are initially admitted at intervals determined by Equation 4.1, this spacing will be destroyed if routers have queue sizes bigger than one packet. Routers should attempt to preserve spacing if possible, and refrain from reordering packets. In the simulations presented in Chapters 8 and 9, First-Come-First-Served queueing was used, with no attempt to preserve packet spacing. Section 10.10 shows that, despite this, packet spacings are preserved.

4.6 Source Error Control Function

Any function that provides the error control desired by the transport protocol can be used here, as long as it meets the design decisions. That is, it must be orthogonal to the protocol's flow control, it should not retransmit data unnecessarily, and it should make no reliance on round-trip timers or round-trip time estimation if possible. Chapter 5 discusses a transport protocol which meets these requirements using Selective Retransmission.

It should be noted that, as shown in Figure 4, any packet retransmissions will form part of the uniformly-spaced packet stream exiting the source's Packet Admission Function.

4.7 Destination Acknowledgment Function

As outlined in the previous section, a transport protocol should not retransmit data unnecessarily. The destination's Acknowledgment function must be designed with this requirement in mind. Thus, Go-Back-N is ruled out, and Acknowledgment functions such as Selective Ac-

knowledgment should be used. The destination should return acknowledgments to the source as soon as possible, in order to provide timely feedback of the sustainable rate values. Where an acknowledgment acks several data packets, only the *most recent* sustainable rate value should be returned, as this holds the most timely sustainable rate information from the network.

4.8 Router Sustainable Rate Measurement Function

The sustainable rate measurement function in each router is the cornerstone of the proposed congestion control framework. A discussion on the requirements and goals of the function, and an example instantiation, is presented in Chapter 6.

4.9 Router Packet Queueing Function

The router may use any packet queueing function which meets the design decisions. Queueing functions which attempt to preserve packet spacings for each source may be useful. First-Come-First-Served was used in the simulations described in Chapters 8 to 11.

4.10 Router Packet Dropping Function

Again, the router may use any packet dropping function which meets the design decisions. Dropping functions which attempt to preserve packet spacings for each source may be useful. If the rate-based framework keeps the network operating at maximum power, packet loss will occur rarely, and the choice of functions for both Packet Queueing and Packet Dropping will have little effect on the framework. This is borne out in the simulations described in Section 11.8. Drop Tail was used in the simulations described in Chapters 8 to 11.

4.11 Router Congestion Avoidance

As noted previously, congestion avoidance schemes attempt to keep the network operating at the knee of Figure 2 (pg. 4). As the main source of congestion information in the framework is the sustainable rate measurement, the router can lower the **Rate** fields in packets further than normal, if it believes it is operating above the knee of the curve. In doing this, it must set itself as the **Bottle_Node**. This extra lowering of the **Rate** field will cause the source to throttle its transmission, and help to bring the router back below the knee of the curve.

4.12 Router Congestion Recovery

Although the congestion framework has congestion avoidance, there may be times when a router reaches the point of congestion collapse. For example, a new traffic flow may learn about its rate allocation before other large round-trip flows learn that their allocation has been reduced. For a small period of time, more packets will be admitted into the network than can be transmitted.

The large round-trip flows will eventually learn about their new rates, but only after one round-trip. It should be possible to inform these flows about the problem in less than one round-trip. This control mechanism must also not overburden the network with congestion control information.

I propose as the congestion recovery mechanism a variant of Source Quench. When a router considers itself congested, it returns **Rate Quench** packets to the sources it believes are the cause of the congestion. These packets also have a **Return Rate** field, which the congested router sets to values which will alleviate congestion from each of the congesting sources. Upon receipt of a **Rate Quench** packet with a **Return Rate** field lower than the source's current rate, the source is obliged to immediately lower its rate to the value in the **Rate Quench** packet. **Rate Quench** packets can never raise a source's transmitting rate.

In order to ensure stale **Rate Quench** packets are not used, a source must discard **Rate Quench** packets which have a sequence number less than the last valid **Rate Quench** packet received.

A policy on **Rate Quench** packet generation is discussed in Section 8.4.

One problem with Rate Quench congestion control is that all routers along a flow's path may send Rate Quenches if a flow's rate is lowered. This is an excessive amount of control information admitted into the network. Section 11.3 describes how the use of Rate Quench packets affects the congestion control of the framework.

4.13 Summary

The congestion control framework outlined in this chapter is very different from traditional schemes used in connectionless packet-switched networks: sliding windows are not used, round-trip time estimation is not used, Go-Back-N acknowledgment is not used. The bulk of the work in the framework is now performed by the network routers, which are in an ideal position to observe network congestion and take measures to alleviate it.

The Transport Layer is reduced in complexity, with a Leaky Bucket-style flow mechanism controlled by the sustainable rate measurements returned (via acknowledgments) from the routers.

Any transport protocol which meets the requirements described in this chapter may be used in the framework: for example, NETBLT, VMTP and XTP could be used. As much of the functionality of these protocols was not required in the rate-based congestion control framework, I chose to develop a new rate-based transport protocol, TRUMP. The description of this protocol is given in the next chapter.

The sustainable rate measurement function in each router plays a central role in the framework. Its requirements and goals, and a specific algorithm implementing these, RBCC, is discussed in Chapter 6.

Chapter 5

TRUMP

This chapter outlines the structure of TRUMP[1] a new message-based transport protocol designed by the author to meet the requirements for a Transport Layer given in the previous chapter. As well, TRUMP has been developed to be a fully-fledged protocol in its own right. TRUMP has other features not required by the framework. A partial verification of the protocol is described in Appendix B, and a possible set of headers to use TRUMP within an IPv6 network is given in Appendix C.

TRUMP's features are:

- Error detection and recovery using selective acknowledgment and retransmission.
- Explicit and implicit connection setup and connection teardown.
- Rate-based flow control, using information passed to TRUMP by the Network Layer.
- An infinite amount of in-transit data: the source can transmit all of a message before any of the acknowledgments have arrived. This aids high throughput.
- Use of protocol timers which are not based on the round-trip time.
- Up to 16 possible types of transport header/data encryption. The encryption covers all fields except for the protocol version number, the encryption type and the segment checksum.

TRUMP is a lightweight transport protocol that transports messages from one communication endpoint (the *source*) to another (the *destination*). This simple unidirectional message passing[2] allows higher layers to provide bidirectional message passing, remote procedure calls and streams (i.e virtual circuits).

TRUMP does not need an explicit call setup; this is to avoid the overhead of call setup in Distributed Systems where the message size is often small. Instead, call setup can be implicit: here every datagram sent from the source to the destination contains data which is part of the message. Thus, the minimum packet exchange to pass a message is 2 packets: one from the source containing the data, and the other from the destination containing an acknowledgment. Each packet is acknowledged: however, acknowledgment packets may acknowledge more than one packet.

[1] a reshuffled acronym standing for **R**eliable **U**nicast **T**ransport **P**rotocol.

[2] Protocols such as TCP and TP4 associate two data streams between a pair of connected endpoints; one in each direction.

Explicit connection setup and teardown is also provided by TRUMP. This allows negotiation of connection privileges and initial flow control values.

Because TRUMP transmits unidirectionally, the following sections discuss the protocol from the point of view of a *source* sending a message to a *destination*.

5.1 Features of TRUMP

5.1.1 Implicit and Explicit Connection Setup and Teardown

TRUMP allows both implicit and explicit connection setup and teardown. For small messages (such as client-server interactions), implicit connection setup is employed: the source begins transmission of the data packets containing the message; the destination creates the data structures needed to reassemble and deliver the message after reception of the first data packet. If the destination cannot accept the message for any reason, it returns a special acknowledgment packet indicating failure of the implicit connection. Otherwise, it begins to return acknowledgments of successful packet reception.

With explicit connection setup, the source initiates the connection by transmitting a connection setup packet to the destination. The destination returns an acknowledgment to this packet permitting or denying the connection. The destination's acknowledgment should also return initial flow control data that the source can use for data transmission. When a successful connection acknowledgment arrives, the source may commence transmission of the message.

Implicit connection teardown is available in TRUMP: when the last acknowledgment for a message arrives at the source, both the source and destination assume that the connection is over. The destination implementation may retain the message's data structures for a period in order to catch any duplicate or delayed packets from the source.

Explicit connection teardown can be selected by the source, or by the destination if explicit connection setup was used. If chosen by either side, the source must acknowledge the last acknowledgment for a message from the destination; this informs the destination that the connection has been terminated by the source.

5.1.2 Connection Identifiers

During connection setup, TRUMP negotiates a connection number for the connection, and a segment sequence number for the first packet. The connection number uniquely identifies the connection[3] between the source and destination. While not currently used, future versions of TRUMP may use this field as an aid to perform CSLIP-style header compression [Jacobson 90].

Messages passed to TRUMP are segmented, with each segment forming the payload of a TRUMP packet. The segments are numbered incrementally, with the first segment sequence number chosen randomly by the source. With a sequence space of 32-bits, this allows a message to be composed of over 4 billion segments.

[3] i.e uniquely identifies the connection for both the source and destination.

5.1.3 Selective Retransmission

Error detection and recovery is performed using *selective retransmission*. TRUMP messages are dynamically broken into *segments*: each segment is transmitted in a TRUMP packet. The last segment of a message is marked as such.

The destination can acknowledge up to 16 segments, indicating which segments have been lost. Lost segments are retransmitted by the source, along with previous segments whose acknowledgments have not been received from the destination. The latter is used to sustain throughput, as sending a query packet to 'obtain' lost acknowledgments would delay the source by one round trip time.

5.1.4 Qualities of Service

TRUMP provides two qualities of service: 'time-sensitive' communications and reliable communications. With both qualities, a TRUMP destination reassembles the message from its constituent segments, reordering segments and discarding duplicates as necessary.

A TRUMP source retransmits lost segments only for messages being sent in the reliable mode. This implies that for time-sensitive communications, the destination may receive a correctly-ordered message with sections missing. Here the destination can pass the segments received 'as is' to upper communication layers.

The characteristics for each service quality are shown in the following table.

Service Quality	Destination ACKs	Source Retransmits	Incomplete Messages
Time-Sensitive	Yes	No	Yes
Reliable	Yes	Yes	No

5.1.5 Dynamic Fragmentation and MTU

Messages are not totally segmented before transmission, as the Maximum Transmission Unit of the underlying network between the source and destination may vary during the message transmission. Instead the transport protocol keeps a current *MTU* value for the underlying network between the source and destination, and uses this to dynamically segment the message. This MTU can be updated from information passed on by the Network Layer.

The MTU may decrease during transmission. In this case, several transmitted segments may not be received because the exceeded the new MTU. TRUMP is able to break a segment into 2 to 16 *fragments*, and retransmit them as new segments. TRUMP's headers include enough information to allow the original segment to be reassembled from the fragments.

5.1.6 Flow Control and Throughput

TRUMP employs rate-based flow and control to maintain its rate at a value which can be sustained by both the destination and the underlying network. Flow and congestion control is also separated from error detection and recovery. Data is transmitted continuously, rather than stopping transmission to wait for acknowledgments which will take one round trip time to arrive, and will slow down throughput.

Flow and congestion control is performed by a variation of Leaky Bucket [Turner 86] which has unlimited capacity (i.e, TRUMP does not discard data from the application layer): the inter-packet delay is set from the congestion information supplied by the network according to Equation 4.1. The Leaky Bucket ensures that the bit rate of the transmission is even.

If the source increases its inter-packet delay, it ignores network congestion information in acknowledgments for packets transmitted before the delay change. This allows one round-trip for the effect of the delay change to be measured by the network.

Flow and congestion control is performed on a source/destination basis, and a source may use the transmission delay interval for packets to one destination to transmit packets to other destinations, if the delay is great enough. The Leaky Bucket flow control mechanism would typically be implemented by a 'callout' system [Vahalia 96], where one timer controls the transmission of the next packet in a queue of packets ordered by desired transmission time.

In the situation where explicit call setup is not performed, the determination of the initial inter-packet interval is left unspecified.

5.1.7 Unacknowledged Data

Unlike traditional transport protocols such as TCP or TP4, there is **no** protocol-defined "window size", i.e. no limit on the amount of unacknowledged data at the source. Theoretically, all segments of a large message could be sent before any acknowledgment packets arrive from the destination. Any limit on the amount of unacknowledged data is therefore implementation-dependent.

5.1.8 Integration with Traditional and New Communication Systems

TRUMP has been designed to integrate with both traditional and distributed communication systems. In order to achieve this, TRUMP's headers allow communication endpoint identifiers to be 0, 16, 32 or 64 bits in size. A TRUMP implementation can choose to support one or more of these identifier sizes.

5.1.9 Security

TRUMP's headers provide for the encryption of both the data and the transport protocol headers themselves. Currently, two encryption systems are defined: no encryption, and per-segment DES block encryption. Fourteen other encryption options can be added in the future. A TRUMP implementation can choose to support one or more of the defined encryptions. Key negotiation and management are not covered by the TRUMP specification.

The size of the initial segment sequence number, and its pseudo-random choice by the source also makes connection setup spoofing very difficult.

5.2 Connection Setup

The following sections describe the operations performed as TRUMP goes through the stages of connection setup, data transmission and connection termination. State diagrams which out-

line the operations involved in peer-to-peer communication for each stage can be found in Section B.5.

TRUMP provides both implicit and explicit setup of data exchange connections. A connection is explicitly set up when a source sends a **SETUP** packet, which is answered by a **SACK** packet from the destination, as in Figure 7.

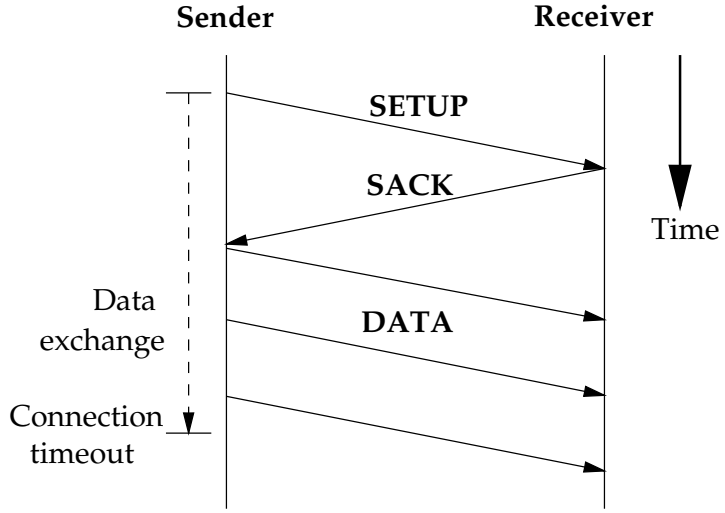


Figure 7: Explicit Connection Setup

The **SETUP** packet holds the connection number for the connection, the initial segment sequence number, the desired quality of service, and the identifiers of the transport-level source and destination communication endpoints. Explicit connection teardown can also be selected by the source or destination with fields in each of the **SETUP** and **SACK** packets.

The **SACK** packet indicates to the source if the connection has been successful, and if not, why not. If the connection is unsuccessful, the source must not transmit any further packets for the connection.

If connection setup is not chosen by the source, implicit connection begins with the transmission of data. Here, the first **DATA** packet is marked as being an implicit **SETUP** packet, and holds the fields mentioned above. When the **DATA** packet is not an implicit **SETUP** packet, the QOS field is invalid.

During connection setup, a destination can discard data packets for the connection before the implicit/explicit **SETUP** packet; this may happen when packets arrive out of sequence.

During implicit or explicit connection setup, the source may set a timer to indicate whether or not the destination is reachable. The value of the timer is implementation-dependent and not specified by the TRUMP protocol.

5.3 Data Transmission

Data transmission in TRUMP is characterised by a source sending a stream of packets to the destination, which returns acknowledgment packets back to the source for a number of data packets, as shown in Figure 8.

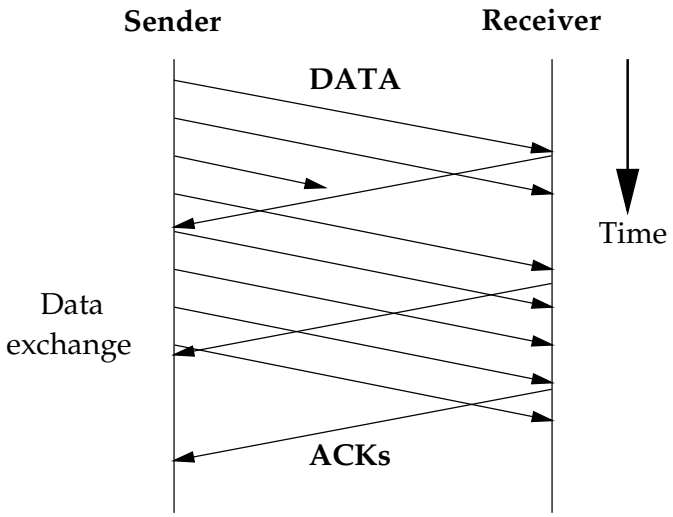


Figure 8: Data Transmission

The transmission of data can be divided into those operations performed by the destination, and those performed by the source.

5.3.1 Destination Operation

The destination is totally driven by packets received from other machines. It must check the checksum on received packets, acknowledge data segments once they arrive, reorder segments and reassemble the message from them.

Not including retransmission, segments are transmitted with monotonically increasing *segment sequence numbers*. Acknowledgment packets hold acknowledgment bitmaps of 1 to 16 bits; thus TRUMP can can acknowledge up to 16 segments in each ACK packet. If bit $G[x]$ is on, the segment with sequence number x was received successfully, otherwise it was not received.

Two sequence numbers are returned with the bitmap; B is the sequence number of the first bit in the bitmap, and T is the highest valid sequence number in the bitmap. Note that $B \leq T \leq B + 15$.

The TRUMP protocol allows the destination to return acknowledgment bitmaps of up to 16 bits, but this maximum is not always achievable. On a real network, a destination will get duplicate and out-of-order packets. The destination operates as follows:

- If a packet with sequence number S arrives such that $S < B$ or $S > B + 15$, then it cannot be acknowledged within the existing partially-filled acknowledgment bitmap. Therefore, the existing acknowledgment, with B, T and the bitmap G, is returned to the source immediately, and a new acknowledgment is created as follows.
- If a packet with sequence number S arrives, and there is no existing acknowledgment bitmap, one is created with $B \leftarrow S$, $T \leftarrow B$, $G[S] \leftarrow 1$ and all other bits in the bitmap turned off.
- If the above two conditions are false, $G[S] \leftarrow 1$, and $T \leftarrow S$ iff $S > T$.
- If $G[S]$ was already set on, an acknowledgment is returned immediately with B, T and the bitmap G.

Finally, the data segments are reassembled by the destination. According to the negotiated quality of service, fully or partially received messages are passed to higher protocol layers.

5.3.2 Source Operation

The source is driven by:

- Messages passed to it from higher layers,
- Acknowledgment packets from the destination,
- Sustainable rate information provided by the Network Layer, and
- Timeouts on unacknowledged data.

Messages passed from higher layers to a TRUMP source specify the source and destination communication endpoints (and network addresses), and the quality of service required. If the source has no initial sustainable rate or maximum transmission unit about the network to the destination, it may be able to ask the Network Layer to provide this information. This mechanism is left unspecified in the TRUMP protocol. Alternatively, an explicit connection may provide the source with this information.

The source then begins to segment the message. Each segment is transmitted, with the first octets of successive packets separated by the current inter-packet delay, to limit the throughput to that sustainable by the network and the destination.

Fragmentation and transmission continues until there is no new data from the higher-level application to send to the destination. Here, the source may choose to retransmit outstanding segments, in order to prompt the destination. These should be transmitted cyclically in their original transmission order, spaced by the current inter-packet delay for the connection.

Acknowledgment packets from the destination:

- Possibly update the inter-packet delay,
- Possibly update the maximum transmission unit, and
- Cause selective retransmissions to occur, if needed.

A TRUMP source keeps three linked lists of segments:

- The **lost** list holds the list of segments known to have been lost,
- The **not-lost** list holds the list of segments in transit, but not known to be lost, and
- The **not-sent** list holds the list of segments not yet sent by the source.

Segments marked as lost in an acknowledgment are moved to the tail of the lost list. The choice of which segment to transmit next is as follows:

1. If the lost list is not empty, choose the segment at its head, transmit it, and move it to the tail of the not-lost list.
2. If the lost list is empty and the not-sent list is not empty, choose the segment at the head of the not-sent list, transmit it, and move it to the tail of the not-lost list.
3. If both the lost and not-sent lists are empty, choose the segment at the head of the not-lost list, transmit it, and move it to the tail of the not-lost list.

4. If all three lists are empty, then no packets are available to transmit. If the application has requested that the connection be closed, TRUMP can perform connection teardown.

Note that when a time-sensitive quality of service is chosen, no segments are placed on the lost list, and it is always empty.

The third choice above causes TRUMP to lazily retransmit in-transit segments until they have been acknowledged by the destination. This mechanism prompts self-clocking acknowledgments from the destination, and avoids the use of a round-trip timer in the TRUMP source. Although this unnecessary retransmission of data goes against the design requirements of the rate-based framework, it does prove that no round-trip timer is required to perform error recovery.

Lazy retransmission can be replaced or augmented with a very coarse-grain round-trip timer, to prompt the source to retransmit some segments if the destination does not respond. Accurate round-trip estimation is not required.

5.4 Connection Teardown

The last packet of data from the source is marked as such, and indicates the end of data. Data exchange is considered to be complete if the destination has received all data segments, and the source has received acknowledgments for all data segments.

At the point when a TRUMP destination implementation has all portions of a message, it sets a close timer which is used to detect further packets for the connection: these might have been caused by a loss of acknowledgments from destination to source. The destination should acknowledge further packets from the source to complete the data exchange. Once the close timer expires, the destination assumes data exchange is complete, and can release its internal data structures as required. This is an implicit connection teardown. The value of the timer is implementation-dependent and not specified by the TRUMP protocol.

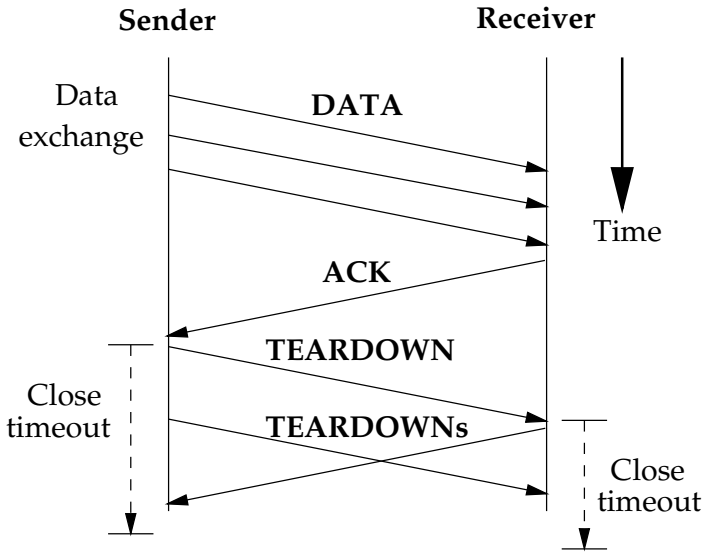


Figure 9: Explicit Connection Teardown

If explicit connection teardown was requested by either side and once the source has all acknowledgments, it sends a **TEARDOWN** packet to the destination. If/when this is received, the destination can formally close the connection as if the close timer had expired. It returns the **TEARDOWN** packet back to the source as well. This is shown in Figure 9.

The source is free to transmit **TEARDOWN** packets periodically, to ensure that the destination receives one. It must not exceed the value of the last **Return_Rate**. The destination must return all received **TEARDOWN** packets back to the source.

The source is also free to set an implementation-dependent close timer during **TEARDOWN** packet transmission. If the timer expires before a **TEARDOWN** packet is received from the destination, the source can assume that the connection is successfully closed.

5.5 Abnormal Connection Termination

At any stage during the exchange of data, a destination may abnormally terminate the exchange and close the connection. This is done by returning to the source a **SACK** packet indicating the reason for connection termination. All new packets from that connection will then be discarded by the destination. Upon receipt of the **SACK** packet, a TRUMP source discontinues the data exchange, marks the connection as closed if necessary, informs higher communications layers of the error, and discards any buffers and data structures used during the connection.

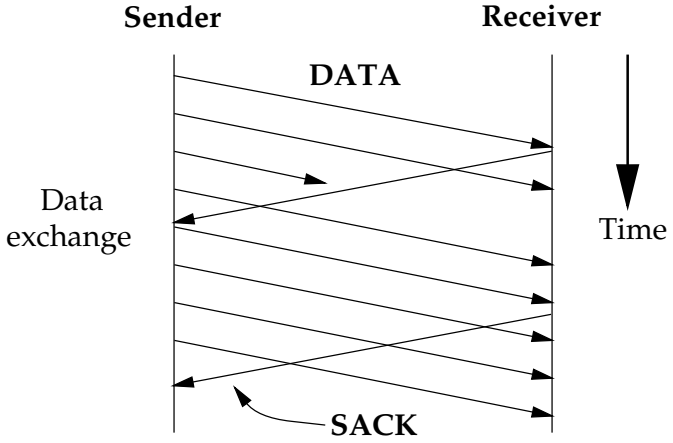


Figure 10: Abnormal Connection Termination

A destination implementation is permitted to retransmit the **SACK** packet several times if data packets for the connection continue to arrive. Source implementations must cope with the receipt of several **SACK** packets for the same connection.

5.5.1 End-to-End Flow Control

As described, TRUMP provides no end-to-end flow control. A sliding window flow control mechanism could be utilised, but this would be undesirable in view of the design requirements of the rate-based congestion control framework. A rate-based form of flow control would suit TRUMP better.

In every TRUMP header returned by the destination to the source, a **Flow_Rate** field holds the highest bit rate for which the destination can accept TRUMP packets from the source. The TRUMP source must use the most recent **Flow_Rate** field, and must not exceed this rate in the

transmission of TRUMP packets to the destination.

The mechanism which sets the **Flow_Rate** field in the destination is not specified. The destination may use such techniques as Multiplicative Decrease/Additive Increase, or it may use techniques which estimate the rate of input buffer draining by applications. The source remains oblivious to the mechanism employed.

Note that a TRUMP source must obey both the **Flow_Rate** set by the destination, and the **Return_Rate** set by the Network Layer. In practice, the minimum of the two must be used. As well, a TRUMP source must not advertise a **Desired_Rate** higher than the most recent **Flow_Rate** value. This ensures that network capacity is not wasted when a flow is bottlenecked by its destination.

5.6 Summary

TRUMP, as a transport protocol, has been designed not only to provide the rate-based flow control as required in the proposed congestion control framework, but to provide other services and features useful to applications. This chapter has described most of TRUMP's features, including those not relevant to congestion control.

The next chapter resumes the exposition of the rate-based framework with a discussion on the central element of the framework, the Sustainable Rate Measurement function which exists in every network router.

Chapter 6

RBCC – Measuring a Source's Sustainable Rate

One of the critical components of the proposed congestion control framework is the Sustainable Rate Measurement function. It has the task of determining a set of rates for all traffic flows in the network so as to prevent any router from becoming congested. As each router in the network has an instantiation of this function, each instantiation must communicate with the others to find this set of rates; therefore the function must work correctly in a distributed fashion.

In this chapter, I will outline the requirements for the Sustainable Rate Measurement function, examine an algorithm called RBCC[1] which meets these requirements, and outline how RBCC works within the congestion control framework.

6.1 Requirements of The Sustainable Rate Measurement Function

In order to perform its task correctly, the Sustainable Rate Measurement function should meet the following requirements:

- The function should allocate enough bandwidth to meet each traffic flow's desired rate, if possible. This ensures that each source's throughput is as high as possible, and network utilisation is high.
- The function should allocate bandwidth to flows so that no output interface's available bandwidth is exceeded. This keeps the incoming traffic down to a rate which can be re-transmitted.
- A router is a **bottleneck** if the sum of the **Rate** fields for traffic flows crossing an interface exceeds the available bandwidth of that interface. If a router is a bottleneck, the function should communicate its allocated rates to each source affected, and to other routers. Thus, the framework adjusts to take into account the bottlenecks in the network.
- The function should allocate bandwidth to flows, taking into account any bottlenecks upstream (towards the source) and downstream (towards the destination). This is a corollary to the previous point.
- Flows should be free to change their desired rate at any time and to any value. The function should cope with changes in a traffic flow's desired rate.

[1] Rate Based Congestion Control

These requirements are mandatory. Other highly desirable requirements are:

- The function should allocate bandwidth fairly amongst a mix of traffic flows with infinite and finite desired rates.
- The function should treat flows with high round-trip times and a high number of intermediate routers equally to flows with low round-trip times and a low number of intermediate routers.

6.2 RBCC — An Example Function

For many of the proposed framework's functions, existing mechanisms can be used to perform the function. This is not true for the Sustainable Rate Measurement function. To overcome this, I have designed an algorithm called RBCC which satisfies all of the requirements given above for the function.

RBCC is implemented as part of the operations performed in every router. When a packet reaches an RBCC router, it performs the following operations:

1. Receive the packet.
2. Determine the output interface on which the packet must be retransmitted (i.e perform routing).
3. Decide to update any static congestion information based on the values of the congestion fields in the packet.
4. Decide to change the **Rate** and **Bottle_Node** fields in the packet based on the static congestion information.
5. Queue the packet on the output interface for transmission by the Link Layer, or drop the packet if there is no room in the output queue.

Operations 1, 2 and 5 must be done in any router, as was seen in Chapter 1. Operations 3 and 4 are part of the RBCC scheme. To these two we can add two more RBCC operations: the actual update of the static congestion information and the actual change of the congestion fields in the packet. An overview of the RBCC algorithm, and the data required for it to perform its work, is show in Figure 11. Also shown are the operations normally performed by a router.

The data required by RBCC to work are the packet's congestion fields and some static congestion information. The latter is known as the Allocated Bandwidth Table, and is described in the next section.

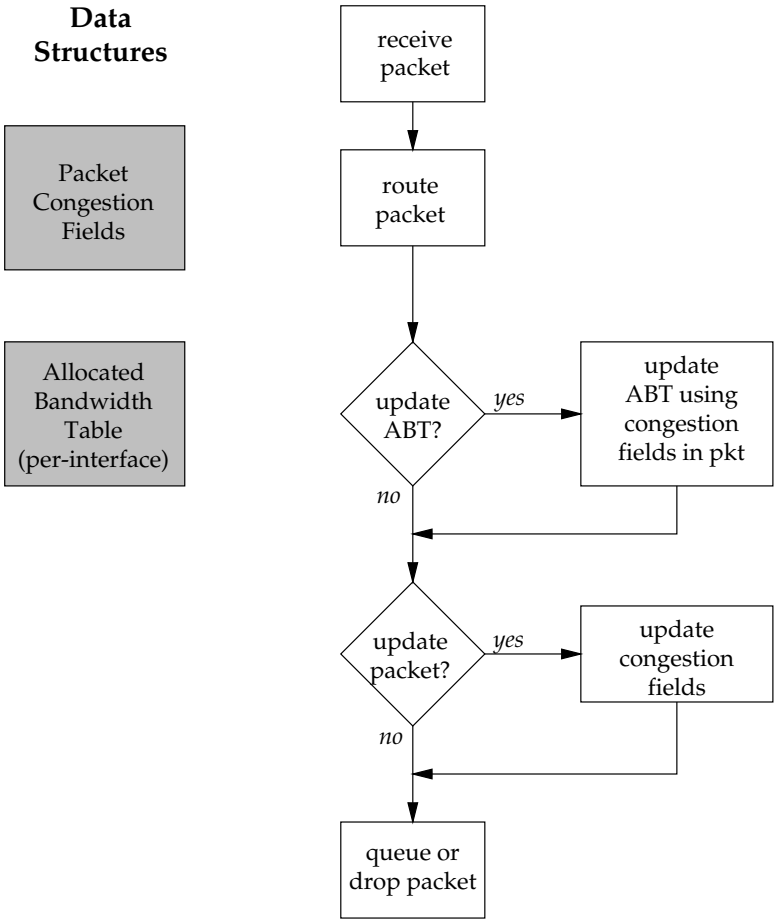


Figure 11: Simplified Execution Flow Diagram for RBCC

6.2.1 The Allocated Bandwidth Table

In order to calculate sustainable rates for traffic flows, RBCC needs to maintain information about these flows. Each router has a number of output interfaces which connect it to other routers, or to sources and destinations of data. For each output interface, RBCC keeps a table of information which describes the bit rate that has been allocated for each flow of data currently passing through the output interface. This table is known as the *Allocated Bandwidth Table*, or the **ABT**. Each entry in the table has the following fields:

Flow_Id, a value which uniquely identifies this flow of data.

Desired_Rate, the bit rate which the source of this traffic desires; it can be ∞.

Limiting_Rate, the limiting rate for the flow, described in Section 6.2.3.

Rate, the bit rate which has been allocated by the router for this flow of traffic; it cannot be zero or a value greater than the overall bit rate of the output interface. Similarly, the sum of the **Rate** fields in the table must not exceed the overall bit rate of the output interface.

Bottle_Node, the router which this router believes is the bottleneck on this flow of traffic. It may or may not be the current router.

A discussion on the **Flow_Id** is delayed until Section 7.2. The next sections describe the four operations performed by RBCC, and how the packet fields and the Allocated Bandwidth Tables

are used by RBCC, with reference to Figure 11.

6.2.2 Deciding to Update the ABTs

The first operation in RBCC is to decide if the Allocated Bandwidth Tables need to be updated. This decision is made heuristically, and is performed using the congestion fields in the newly-arrived packet and the flow's ABT entries.

The Allocated Bandwidth Tables do not need to be updated on every packet arrival; in many cases, the packet contains the same congestion information as the last packet for this traffic flow. An optimisation for RBCC would be to cache the last-received congestion fields for every traffic flow; if the newly-arrived packet's fields are the same as the cached fields, the ABTs do not need to be updated. This optimisation is omitted below for clarity, but is performed in the simulations described in Chapters 8 to 11.

For every packet received by a router, there are *two* ABTs which could be updated. The first ABT is for the output interface to which the packet is destined. The second ABT belongs to the interface which retransmits packets coming in the reverse direction (i.e destination to source). Only the **Return_Rate** congestion field is used to update the second ABT. RBCC decides to update these ABTs on the conditions outlined below.

The first/forward ABT can be updated when:

- The packet is for a new traffic flow through the output interface. This new traffic flow must be allocated some bandwidth by the router.

  The ABT **Desired_Rate**, **Rate** and **Bottle_Node** field are set to the packet's values. A **Flow_Id** is created which identifies the new traffic flow.
- The **Desired_Rate** field in the packet differs from the **Desired_Rate** field currently in the ABT. The router may need to lower its allocated rate so as not to exceed the new **Desired_Rate**; alternatively, the router may be able to allocate more bandwidth if the **Desired_Rate** has increased.

  The ABT **Desired_Rate** field is set to the packet's value.
- The packet/ABT **Rate** fields differ. This may be due to a change in bottleneck upstream, or a change in the flow's rate allocation upstream. In any case, this router may need to update its own Allocated Bandwidth Table.

  The ABT **Bottle_Node** field is set to the packet's value, and the ABT **Rate** field is set to the packet's **Rate** value.

The second/reverse ABT can be updated when:

- The packet **Return_Rate** field is lower than the ABT **Rate** field. A downstream router has lowered the traffic flow's bit rate, and this router needs to update its ABT to reflect this.

  The ABT **Rate** field is set to the packet's **Return_Rate** value. If the ABT **Bottle_Node** field identifies this router as the bottleneck, then this field is changed to any other value.

Note that the ABT **Bottle_Node** field is only required to identify if the router is, or is not, the flow's bottleneck.

These heuristics ensure that the ABT contains the correct information required to determine the sustainable rates for all traffic flows passing through the output interface. If any ABT fields have been modified, then the appropriate ABT must be updated.

6.2.3 Updating the ABTs

Updating an Allocated Bandwidth Table given the information already in the table and the congestion information in a packet is not trivial. We have considered the conditions which might prompt recalculation of the information in an ABT. Let us now consider the work involved.

There are several criteria which must be taken into account when updating an ABT:

- Allocation of rates amongst the flows in the ABT must be fair.
- A traffic flow's rate allocation must not exceed the desired rate of its source, nor may it exceed the rate set by a bottleneck router which is not this router.
- A traffic flow's rate allocation cannot be set to zero or to ∞.
- The sum of allocated rate in the ABT must not exceed the output interface's overall bit rate.

Rate allocation must be fair, but the ABT may contain flows with many different desired rates; these may be finite or infinite as well. This makes rate allocation difficult. The rule of thumb that I have chosen for RBCC is:

> A flow has a desired rate D and has an existing rate allocation R from its bottleneck router[2]. The flow's *limiting rate* L is D if we are the bottleneck router, otherwise $min\{D, R\}$.
>
> A flow with limiting rate L should receive the same rate allocation A as all flows with limiting rates $\geq L$, unless $A > L$. In the latter case, the flow is allocated L, and the difference, $A - L$, is distributed to the flows with higher limiting rates, according to this rule.

This distribution rule gives the following results:

- Flows with equal limiting rates will be allocated the same bit rate.
- When bandwidth becomes scarce, flows with low limiting rates are more likely to preserve their existing rate allocations, and flows with higher limiting rates are more likely to have their allocations decreased.

In the extreme case when $A < L_{lo}$ for the flow with the lowest limiting rate, *all* flows will receive the rate allocation A. Only when flows cannot use their allocation A is the excess passed on to 'faster' flows. Therefore, the distribution rule treats flows with varying desired and limiting rates fairly.

The ABT update criteria and the distribution rule combine to form the ABT update algorithm used by RBCC. This algorithm is given below in the C language[3].

[2]Note that R may be ∞ when there is no bottleneck router.

[3]This code is abstracted from the file `router/rbcc.c` in the modified REAL network simulator, described in Chapter 8.

```
/* Data Structures */
struct conv_type                        /* Table entry for a single flow */
{
    int               flow_id;          /* Unique identifier for this flow */
    int               bottlenode;       /* Node which is the bottleneck */
    double            rate;             /* Rate allocated by a node */
    double            desired_rate;     /* Rate desired by the source */
    double            lim_rate;         /* Limiting rate on the flow */
    struct conv_type *next;             /* Pointer to next flow in the list */
};


/* Input Variables */
double bandwidth;                       /* Available bandwidth on interface */
struct conv_type *head_of_flow_list;    /* List of flows requiring bandwidth */
                                        /* on the output interface. Note that */
                                        /* this list is ordered by the */
                                        /* limiting rate field (ascending) */
int flowcount;                          /* Number of flows in the list */
int node;                               /* Router's unique identifier */

/* Local Variables */
double bandwidth_left;                  /* Bandwidth not yet allocated */
int flows_left;                         /* Flows with no allocations yet */
struct conv_type *c;                    /* Pointer to each flow in the list */
double allocation;                      /* Bandwidth allocation chosen for */
                                        /* each flow */


/* Output Variables: the rate and bottlenode fields in the linked list */
/* are updated by the algorithm. */


/* Algorithm */

  /* Fairly divide the bandwidth up amongst the flows. */

  bandwidth_left= bandwidth; flows_left= flowcount;
  for (c= head_of_flow_list; c!=NULL; c= c->next) {

                /* The bandwidth_left variable contains the bandwidth */
                /* yet to be allocated to remaining flows. We assume that */
                /* this flow can use an N'th amount, where there are */
                /* N flows remaining to get allocations */
      allocation = bandwidth_left / (1.0 * flows_left);

                /* If the flow's limiting rate is higher, */
                /* we become its bottleneck and set its rate. */
      if (c->lim_rate > allocation) {
          c->rate= allocation; c->bottlenode=node;

                /* Otherwise, the flow is bottlenecked elsewhere. */
                /* Only give it its limiting rate. The excess from */
                /* the allocation calculated above will be shared amongst */
                /* the flows with higher limiting rates. */
      } else {
```

```
            allocation= c->lim_rate; c->rate= allocation;
        }

        /* Subtract the allocation actually given to the flow from the */
        /* remaining bandwidth. */
        bandwidth_left-= allocation; flows_left--;
    }
```

6.2.4 Updating the Packet's Congestion Fields

After the Allocated Bandwidth Tables have been recalculated (if required), there are two instances when a router can change a packet's **Rate** and **Bottle_Node** fields. If the router has allocated a lower bit rate than that indicated by the **Rate** field, then the **Rate** field can be lowered and the **Bottle_Node** field updated to the value in the ABT.

When the router is itself the bottleneck on the traffic flow, the **Rate** field can always be overwritten by the **Rate** field in the ABT. This has the effect of increasing the bandwidth available to the traffic flow when there is excess bandwidth at the router (another traffic flow may have terminated). The **Bottle_Node** field is also updated to be the value in the ABT.

Similarly, in the reverse direction, the **Return_Rate** value can be overwritten if the router is the bottleneck router, as shown in Figure 12.

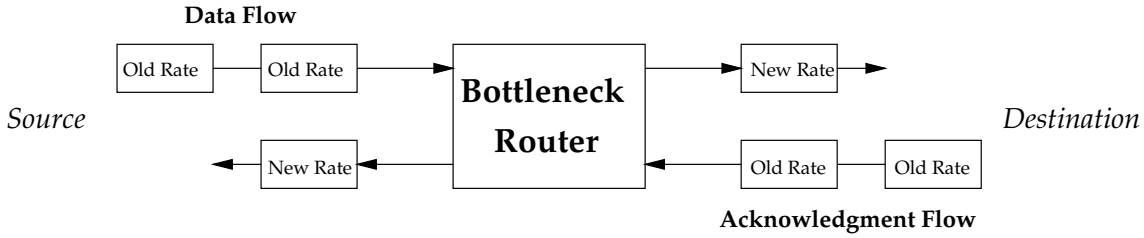


Figure 12: Reverse Congestion Field Update

The bottleneck router has set a new rate for the traffic flow. The **Rate** fields in data packets are updated as they pass through the router. As well, **Return_Rate** fields in acknowledgment packets are also updated. The source learns about the new allocated rate faster than if the **Return_Rate** fields were not being updated.

6.3 TRUMP/RBCC Operation

A rate-based transport protocol such as TRUMP uses the sustainable rate measurements from RBCC in order to minimise network congestion. Let us now look at the operation of TRUMP and RBCC in tandem.

6.3.1 Source Transmission

TRUMP, as a transport protocol, provides not only rate-based flow control, but error correction via packet retransmission and selection of the type of connection required between the source and destination. This section will concentrate on the flow control aspects of TRUMP.

A TRUMP Transport Layer can begin transmission either by performing a two-way handshake with the destination Transport Layer, or by starting immediately with data transmission. The two-way handshake is performed as follows:

1. Transmit connection setup packets to the destination with an inter-packet separation of S seconds.
2. If no setup acknowledgment packet is received after T seconds, abort the connection. Stop.
3. If a setup acknowledgment packet is received, discontinue setup packet transmission and start transmitting data packets.

In the simulation of TRUMP and RBCC, S is 1 second and T is ∞. In reality, T would be finite.

During data transmission, TRUMP transmits data packets spaced by an inter-packet delay, whose value is calculated using Equation 4.1, which is given again below:

$$delay = \frac{packet\ size\ in\ bits}{rate\ in\ bits\ per\ second}$$

where $rate\ in\ bits\ per\ second$ is obtained from the **Return_Rate** field from either the connection setup acknowledgment packet or the last received data acknowledgment packet. Remember that an increase in the delay causes TRUMP to ignore **Return_Rate** fields for one round-trip. A single timer per connection is set to the inter-packet delay, and this prompts the TRUMP source to transmit the next data packet.

As data packets pass through the network, their congestion fields are (possibly) set by the intermediate routers according to the RBCC algorithm.

6.3.2 Packet Reception by the Destination

After being forwarded by all the routers between the source and destination, and having the **Rate** and **Bottle_Node** fields set to reflect the bandwidth available to the corresponding traffic flow, a data packet reaches the destination. This may or may not prompt the transmission of a selective acknowledgment. In any event, a selective acknowledgment is eventually transmitted to the source, and its **Return_Rate** field are set to the values received in the most recent packet.

As the stream of selective acknowledgments forms a traffic flow in its own right, the **Desired_Rate** field in each acknowledgment must be set to a sensible value. The method used to determine the desired rate for the flow of acknowledgments back to the source is:

$$desired\ acknowledgment\ rate = most\ recent\ data\ rate\ value * \frac{ack\ packet\ size\ (in\ bits)}{total\ data\ being\ acked\ (in\ bits)}$$

This has the effect of limiting the desired rate of acknowledgment packets to a fraction of the available bandwidth for data packets, with the fraction based on the ratio between acknowledgment packet sizes and the total size of data packets being acknowledged.

The assumption behind the equation above is that the data traffic flow is the limiting factor on the acknowledgment traffic flow. In some circumstances, it may be the other way around: this is not yet catered for.

In the simulation of TRUMP, the transmission of acknowledgment packets are not explicitly separated with any inter-packet delay; it is assumed that the spacing of data packets from the source will correctly space the acknowledgment packets from the destination. The results of this assumption are described in Section 10.10.

6.3.3 Packet Reception by the Source

As with the data packet, the acknowledgment packet is forwarded by several routers before being received at the source. As with the destination, the values of the **Rate** and **Bottle_Node** fields received by the source are returned to the destination as the **Return_Rate** field field in future data packets.

The **Return_Rate** field from the acknowledgment packet is used by the source to recalculate the inter-packet delay, using Equation 4.1 as before:

$$delay = \frac{packet\ size\ in\ bits}{Return\_Rate\ in\ bits\ per\ second}$$

The new delay takes effect after the source's next data packet transmission.

6.3.4 Flow Termination

Correct calculation of a router's ABTs depends upon the router knowing about all the traffic flows through it. It can deduce the start of a new traffic flow by its packets, but it cannot easily deduce the end of a traffic flow from a lack of packets; the flow may have been slowed down by a bottleneck somewhere.

To remove this problem, two more congestion control fields must be added to the four described in Section 4.4. These are:

Flow_Id: A value which uniquely identifies this flow of data. Strictly, the concatenation of the source address and the **Flow_Id** is unique.

Flow_End: A boolean which indicates that the flow has ended.

Transport Layer protocols such as TRUMP must ask the Network Layer to set the **Flow_End** flag for packets which terminate the flow of traffic. When RBCC receives packets with the **Flow_End** flag set, it removes the corresponding traffic flow entry from the ABT and then recalculates the table. Further packets with the the **Flow_End** flag set have no flow allocation, and so should require a negligible overall bit rate.

A TRUMP source sends connection teardown packets every second once there are no packets left to send, and stops sending packets after it has received one teardown packet back from the destination. All connection teardown packets have the **Flow_End** flag set.

Upon receipt of a *teardown* packet, a TRUMP destination immediately reflects it back to the source, again with the **Flow_End** flag set.

6.3.5 Abnormal Loss of a Traffic Flow

If packets from a traffic flow do stop passing through a router, then unless the corresponding ABTs are recalculated, the bandwidth of the output interfaces will be underutilised. If a flow

terminates abnormally, or if events such as a route change at another router take place, the router will not see packets with the **Flow_End** flag set.

In this situation, all that the router can do is to assume that the traffic flow has stopped after a certain period of time has expired. The optimal value for this time delay would be the inter-packet delay for the traffic flow scaled up by a small factor K; if no packets are seen after this time, then the router can assume that the flow has stopped.

This introduces a dependency on inter-packet delay estimation, which the Sustainable Rate Measurement function as presented does not perform. In the simulation of RBCC in REAL, a constant $F = 5$ seconds was used. This of course implies that once a flow stops, the router will be under-utilised for up to 5 seconds. Alternatively, if a traffic flow spaces its packets with intervals of >5 seconds, the ABTs will be recalculated on receipt of every packet in the flow.

6.3.6 Congestion Avoidance

One of the main factors used to calculate the sustained rate values in RBCC is the actual bandwidth of an output interface, which is constant. As this constant has an overriding effect on the sustained rate values, it can be scaled when a router believes it to be above the knee of the power curve, to provide congestion avoidance.

In my simulations, I chose to scale the interface's bandwidth value as follows:

$$bandwidth\ value\ used = \begin{cases} actual\ bandwidth & \text{if } packets\ queued \leq T \\ actual\ bandwidth * \alpha & \text{if } packets\ queued > T \end{cases} \quad (6.1)$$

In my simulations, values of $T = 5$ and $\alpha = 0.75$ were used, where each router could queue up to 10 data packets, and this worked well as a congestion avoidance scheme.

Other bandwidth scaling possibilities are to have low- and high-watermarks for the number of packets queued, T_{low} and T_{high}, or to scale the actual bandwidth as a function of the number of packets queued. Scaling may be done with or without hysteresis effects.

With any form of sustainable rate scaling, there is the possibility of source rate oscillation with a period of the round-trip time. Any congestion avoidance scheme using sustainable rate scaling must take this into account. With the scaling technique described above, there were minor oscillations observed in the simulations, which had no significant impact on network performance.

A full description of the effect of T and α is given in Chapter 11.

6.4 Summary

The Sustainable Rate Measurement function is the core element of my rate-based congestion control network. In order to perform correctly and fairly, it must meet several requirements which were outlined in Section 6.1. One such function which does meet these requirements is RBCC. It is composed of a number of heuristics and a rate allocation algorithm.

RBCC sets the **Rate** field in each packet based on the information it keeps in a set of Allocated Bandwidth Tables. These tables are updated by the rate allocation algorithm if the heuristics so indicate. The rate allocation algorithm uses congestion information in packets to determine a fair set of sustainable rates for all flows of traffic through a router.

The elements in the rate-based framework, and functional instantiations for two elements, have so far been discussed. The next chapter will examine some of the issues raised during the exposition of the framework. The chapters following will then examine the implementation of the rate-based framework within a network simulator, and the performance of the framework's congestion control.

Chapter 7

Comments on the Congestion Control Framework

The previous three chapters have described my proposed rate-based congestion control scheme in some detail. There have been several issues raised which require further consideration; these include the available parameters of the scheme and its behaviour. I will address these issues in this chapter.

7.1 The Framework as a Feedback System

It should be apparent that the congestion control framework forms a closed loop feedback system with a delay of one round-trip; this round-trip includes the delay in generation of selective acknowledgments. Note, however, that the framework does not use the value of the round-trip. The feedback loop is shown in Figure 13; compare this figure with Figure 4 (pg. 34), which has more detail.

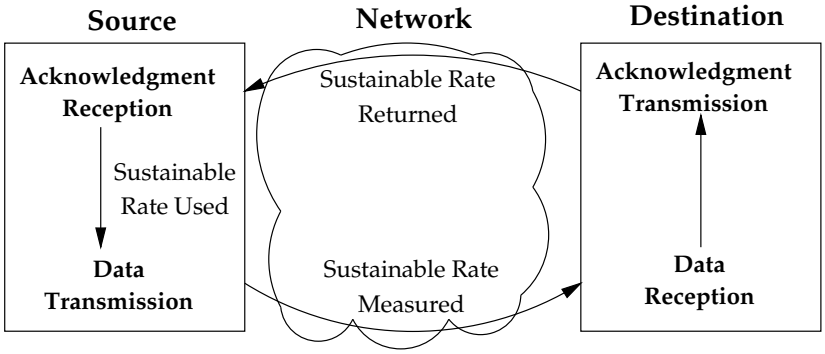


Figure 13: The Framework is a Closed Loop

I have suggested that when the source receives a **Return_Rate** value which lowers its transmission rate, it should not react to further return rate values for at least one round trip time. This allows the effect of the lower source rate to be measured by the network components, and helps to minimise short-term rate fluctuations which would be caused by short-term congestion.

If, on the other hand, the **Return_Rate** value would increase the source's transmission rate, then the network is advertising extra capacity and the source can use this rate value immediately.

In any event, the source's transmission rate cannot go to zero, as this would prevent the source from transmitting due to an infinite inter-packet delay. Similarly, the source's transmission rate cannot go to infinity, as this would cause the source to saturate the network with data. Any Sustainable Rate Measurement function must ensure that the rate returned to a source is neither a zero nor infinity.

When a source begins transmitting, it will not have an initial rate value. There are several alternatives here which are not defined in the TRUMP protocol. If the source has had a similar connection to the same destination in the recent past, it may be able to use the final rate value from that connection. If the source has no historical information, it can start transmitting at a very low data rate. After one round trip, it will have better rate information. If the source performs explicit connection establishment, then it will obtain rate information from the destination before any data has been transmitted over the connection.

The round-trip, and hence the speed of operation of the closed loop, is affected by both the overall link latencies and by the delay in acknowledgment generation. The former is outside of the control of the framework, but a high overall link latency should not penalise a traffic flow. Transport protocols within the framework can control the delay in acknowledgment generation by choosing the number of data packets acknowledged in every selective ack packet: the higher the number, the higher the delay in acknowledgment generation. However, the higher the number acknowledged, the less bandwidth is required for the acknowledgment flow.

It is arguable, in these days of extremely high link bit rates, that the bandwidth required for the acknowledgment flow is negligible. However, if an acknowledgment flow is bottlenecked by a low-speed link, then this is not true. The transport protocol endpoints may examine the sustainable rates and round-trip times for the data and acknowledgment flows and negotiate an appropriate selective acknowledgment size.

In Chapter 11 the effect of different sized selective acknowledgments on the framework's performance is simulated, and the results summarised.

7.2 What Constitutes a Traffic Flow?

The Sustainable Rate Measurement function must uniquely identify each traffic flow, and distribute bandwidth to each traffic flow as a set of sustainable bit rates. We saw in Section 6.2.1 that each entry in the Allocated Bandwidth Tables has an identifier for its traffic flow. But what constitutes a traffic flow? There are several possibilities:

1. The Transport Layer is the source of the traffic flow, acting on behalf of an application. The flow is uniquely identified by the tuple (Source Host, Destination Host, Source Port, Destination Port);
2. The Transport Layer is the source of the traffic flow, which is uniquely identified by the tuple (Source Host, Source Port);
3. The Host is the source of the traffic flow, which is uniquely identified by the tuple (Source Host, Destination Host); or
4. The Host is the source of the traffic flow, which is identified by the tuple (Source Host).

The fourth possibility can be immediately dismissed, as a host may be transmitting to multiple destinations with packets taking multiple routes. The second possibility can be discounted

for transport protocols such as TCP where an application is tied to a particular port, and may be communicating with several destinations from that port. TRUMP also falls into this category.

We are left with options 1 and 3, and each has quite different characteristics. In the first option, every application to application communication is a traffic flow, and there can be several traffic flows between the same source and destination hosts. Assuming the flows all take the same route, each flow is represented as entries in two ABTs[1] in every traversed router. The routers have a set of desired rates with which to divide up the available bandwidth equitably.

In option 3, each source machine to destination machine communication is a traffic flow. Not only does the Transport Layer demultiplex the incoming packet stream to one or more applications, but it must also demultiplex the sustainable rate returned to it amongst the one or more applications communicating with the same destination. It must also sum the applications' desired rates to form a single **Desired_Rate** value; obviously, ∞ plus a finite value is still ∞.

There are several advantages with option 3. There are fewer entries in each routers' ABTs, as the routers perceive fewer traffic flows. The source can choose its own policy on dividing up the available bandwidth to the applications communicating with the same destination, and it is in a better position to determine the applications' real desires than the network. It also requires fewer timers to perform inter-packet delays.

However, there are several disadvantages with option 3. Because application data flows are aggregated, the routers do not perceive the real number of traffic flows, and may distribute bandwidth fairly on a per-source/destination basis but not on a per-application basis. Although the Transport Layer has fewer timers to deal with, it also has to multiplex the applications' data so as to meet the Leaky Bucket's inter-packet delay. This may cause fluctuations in per-application packet spacing, which may be unacceptable for time-sensitive applications.

Finally, different application data flows have different desired qualities of service, and different desired data rate requirements. A file transfer flow desires an infinite and uniform bit rate, but a remote login flow desires a low and spasmodic bit rate. Aggregating different application data flows may penalise flows with certain characteristics and qualities of service.

In the simulations described in the next few chapters, I chose option 1 to identify each traffic flow.

7.3 What Congestion Fields are Required?

If option 1 in the previous section is chosen, each application data flow is a traffic flow in its own right. I have proposed that each packet transmitted in the congestion control framework has the following fields: the **Flow_Id**,**Desired_Rate**, **Rate**, **Bottle_Node**, **Return_Rate** and **Flow_End** fields.

I would suggest that the rate fields can be floating point numbers at least 32 bits in size, and the node field is most likely to be a network node identifier. The **Flow_End** field can be a single bit. The **Flow_Id** field, combined with the network address of the source, must uniquely identify the flow: a good minimum for this field would be 16-bits, to allow for 65,536 traffic flows per source. Therefore, an implementation of the framework in IPv4 and IPv6 would require at least 113 and 209 extra bits of header per IP datagram, respectively.

[1] One for the data flow and one for the acknowledgment flow.

The **Rate** and **Return_Rate** fields must exist in each packet, as no feedback would be performed without these fields. The **Desired_Rate** field is extremely useful, as this allows the Sustainable Rate Measurement function to distinguish between flows that will accept any extra bandwidth available, and those which would not make use of the extra bandwidth.

The **Bottle_Node** field is used by the Sustainable Rate Measurement function to determine where a traffic flow is bottlenecked, i.e by the router currently processing the packet, or by another router. Unfortunately, in RBCC's current form, this field is required: this has been confirmed by simulations. If RBCC cannot tell who is the bottleneck of a traffic flow, it cannot determine if any spare capacity can be donated to the flow.

The overhead of 128 bits for the **Bottle_Node** field in IPv6 is large. However, this field only has to indicate which router is the bottleneck. A suggestion would be for each router to randomly choose a smaller unsigned integer as its 'bottlenode' identifier. If a 32-bit identifier was used, a saving of 96 bits would be made, and the probability of an identifier collision across the route of any traffic flow would be very small.

Given these suggestions, a realistic congestion control header for IPv6 requires 192 bits, and is shown in the following diagram. The **Flow_End** flag is indicated by the letter 'E'. The **Version** field allows the congestion control header to be changed without confusing routers using old versions of the congestion control header.

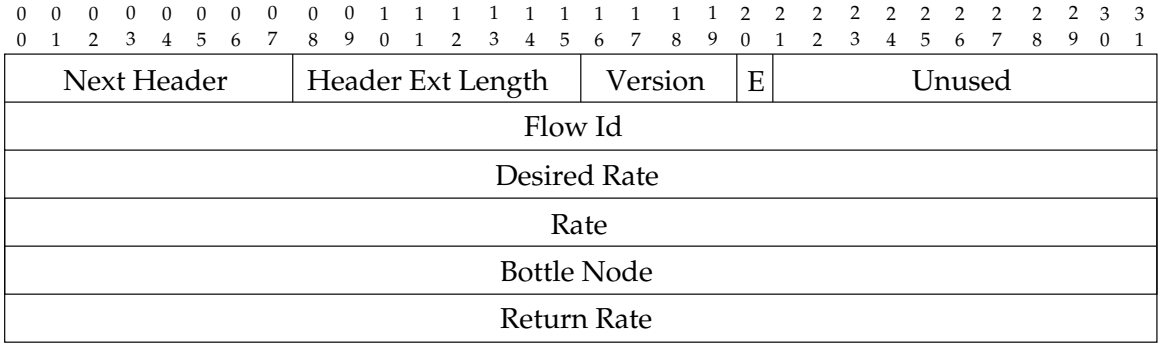


Figure 14: The Congestion Framework Header in IPv6

7.4 Transport and Network Layer Intercommunication

In the proposed congestion control framework, the Transport and Network Layers in the source of each traffic flow must communicate so that the flow is correctly identified and terminated, and that the transmission for the flow occurs at the correct rate. Here is a description of the intercommunication required between the two layers.

1. When a new flow is created by a Transport Layer protocol, a flow identifier must be chosen by the protocol. The flow identifier must uniquely identify the flow within the source.
2. Every Transport Layer segment given to the Network Layer for transmission must be accompanied by the destination network address, a **Desired_Rate**, a **Flow_Id** value, and a **Flow_End** value. If the segment is part of an exchange to terminate the flow, the **Flow_End** value must be true; otherwise, it must be false.
3. After prepending a Network Layer header, and before the Network Layer datagram is transmitted, the Network Layer sets the **Rate** field to ∞ and sets the **Bottle_Node** congestion control field to identify the host. The **Desired_Rate**, **Flow_Id** and **Flow_End** fields are

set to the values from the Transport Layer.

4. Every Transport Layer segment given to a Transport Layer protocol after reception by the Network Layer must be accompanied by the source network address, the value of the **Return_Rate** field and the value of the **Flow_Id** field.
5. When a **Rate_Quench** datagram is received by the Network Layer, the value of the **Return_Rate** and **Flow_Id** fields must be passed asynchronously to the Transport Layer protocol. As well, the Quench must contain the first N octets of the Transport Layer segment from the datagram which prompted the Quench. These N octets must also be passed to the Transport Layer. This allows the Transport Layer to determine which particular segment prompted the Quench.

7.5 Complexity of RBCC

In the proposed rate-based congestion control framework, most of the control burden has been moved into the network's routers. This flies in the face of traditional congestion research which has concentrated on end-to-end solutions. The experimental results in Chapters 9 to 11 show how significantly better the rate-based framework controls congestion, when compared to the end-to-end scheme used by TCP. Unfortunately, routers face an increasing burden from other areas: higher link bandwidths, packet filtering, CIDR routing extensions, IPv6 extensions and sophisticated dynamic routing protocols. Existing Internet routers must route and retransmit up to hundreds of thousands of packets per second, consulting routing tables with tens of thousands of entries.

Therefore, although the proposed rated-based control framework provides significant congestion control benefits, will the extra workload it places on network routers make the deployment of the framework untenable? This issue is addressed briefly here, and in more detail in Appendix D.

The RBCC algorithm must be implemented on every router within a network using the rate-based framework. This entails an extra CPU overhead and an extra memory allocation overhead on each router.

RBCC maintains state about the existing traffic flows across an output interface in the interface's Allocated Bandwidth Table. Many of the operations that deal with the packet congestion control fields must consult this table. Let us look at the complexity of each operation in turn.

During the ABT update decision, the ABT entry corresponding to the incoming packet must be found. A linear search has order of complexity O(N), where N is the number of flows in the ABT. For a new flow, a new ABT entry must be inserted. The entries need to be kept in increasing limiting rate order, and so an insertion sort is required. A linear insertion sort has order of complexity O(N).

During the ABT update proper, available bandwidth must be fairly distributed across all traffic flows in the ABT. With the table ordered by limiting rate, the allocation to each flow is independent of the other flows, and the update operation has order of complexity O(N).

Finally, a packet's congestion fields may need to be updated from the ABT. A reasonable implementation should retain a 'pointer' to the entry from the ABT update decision operation, and this operation should have order of complexity O(1).

The overall complexity of the operations performed by RBCC is O(N), where N is the number of traffic flows represented in the output interfaces Allocated Bandwidth Table.

Without reducing the complexity, the speed of the RBCC operations can be improved by such techniques as caching the most-recently-seen congestion fields for each traffic flow. The ABT does not need to be updated if the cached congestion fields match those in a received packet. The results given in Section 10.11 show that caching appreciably reduces the number of ABT updates.

Similarly, an ABT does not need to be updated if all the flows in the ABT are bottlenecked elsewhere, and the sum of the flows does not exceed the output interface bandwidth. Addition of this pre-update test to the simulated code removed approximately 60% of ABT updates due to cache misses.

Appendix D examines the cost of RBCC on current core Internet routers. The conclusion is that RBCC imposes an extra memory and processing burden on Internet routers which is commensurate with their current memory and packet switching CPU burden.

7.6 RBCC and Fair Share

The RBCC function meets the requirements of the Sustainable Rate Measurement function within the congestion control framework. [Charney 94] describes and analyses a similar algorithm for allocating rates to traffic flows, the 'Fair Share' algorithm.

There are several differences which distinguishes Charney's work from RBCC. In essence, the framework in which Charney's algorithm operates assumes that all traffic flows desire an infinite transmission rate; moreover, each instantiation of the algorithm operates independently of the others. The independence of the algorithm instantiations means that only some of the data which can help to determine optimum data rates is utilised. In RBCC, not only is rate information used in **both** directions, but the identity of the current *bottleneck* for each traffic flow is delivered to each router, which assists in determining the optimum data rates.

Charney notes that finite desired transmission rates can be modeled in the framework by inserting a 'fictitious' link between the traffic source and the first router, where the link has a maximum data rate equal to the source's desired rate. The drawback of this fiction is that the desired transmission rate is thus fixed, and cannot be altered by the traffic source once set.

In Charney's framework, some packets carry a "stamped rate" field, which corresponds to the **Rate** and **Return_Rate** fields in this work, described in Section 4.4. A bit in the packets indicates whether the "stamped rate" field is being propagated to the destination (corresponding to the **Rate** field in this work), or to the source (corresponding to the **Return_Rate** field in this work).

Fair Share does not appear to take into account the bandwidth required by reverse traffic flows (e.g by acknowledgment flows for transport protocols), but the "stamped rate" field for each direction must be carried in separate packets, and it is unclear how the "stamped rate" is returned to the source of acknowledgments. In the framework proposed in this thesis, each packet holds rate fields for both directions: **Return_Rate** information does propagate to the source of acknowledgments.

Sources cannot indicate a desired transmission rate to the Fair Share algorithm. This can cause underutilisation of available bandwidth. This is overcome with an extra bit, the *u-bit*, associated with the "stamped rate" field. "Greedy" flows clear this bit, and flows with fixed

desired rates set this bit on. Intermediate routers set the *u-bit* when their advertised rate for the flow is **less** than that indicated in the "stamped rate" field. If a "stamped rate" field returns to the source with the u-bit off, then the advertised rate of all links was higher than the stamped rate of the original packet. In this case, flows desiring more bandwidth can increase their rate, and the value in the "stamped rate" field, to an arbitrary value. The algorithm will then converge to a new set of optimum values.

Finally, although Charney gives a theoretical analysis of Fair Share, this is done to show that the allocated rates converge to the optimum values, when loads on a network are static, and when they are slowly changing (due both to route changes and the set of traffic flows). The problem of network congestion is not considered, and Charney notes:

> In addition, we allow the switches to drop packets as they please, as long as at least some packets of each session continue to get through. While dropping packets can cause a lot of wasteful retransmissions, it is essential to note that our algorithm will still calculate correct optimal rates even in the presence of heavy packet loss.

The thrust of this thesis is to show that RBCC, an algorithm similar to Fair Share, not only allocates optimum transmission rates, but that network congestion is also *effectively controlled*, when RBCC is used as part of a congestion control framework. An area of future research would be to evaluate the performance of Fair Share within my proposed congestion control framework, and to compare it with RBCC.

7.7 Determination of Desired Rate

There are essentially three types of network applications. The first desire an infinite bit rate: file transfers and the like. The second desire a constant bit rate: voice and video transfers. The third have a distribution of bit rates: remote logins by humans, for example.

The source Transport Layer can easily deal with the first two. The third type of application does not fit so well into the rate-based congestion control framework. One solution is to apply an arbitrary rate limit to an application of this type, at the source Transport Layer.

Assume that the Transport Layer advertises the arbitrary rate limit as the flow's desired rate, and is allocated this desired rate. When the application is transmitting below this rate, extra capacity is reserved but goes unused. When the application is transmitting above this rate, the source transport layer buffers the application data and admits it using the Leaky Bucket scheme into the network.

The difficulty is the choice of the arbitrary rate limit. The source should be able to observe the application's short-term bit rate requirements, and move the arbitrary rate limit to utilise the available capacity as best as possible. Note, however, that the arbitrary rate limit becomes the advertised desired rate, and changes in the desired rate cause extra RBCC work to be performed in the routers along the flow's path. Therefore, a compromise needs to be reached between optimal utilisation and excessive RBCC operations. I have not considered this problem in this thesis.

The rate-based control framework is targeted at long-duration traffic flows. Flows which consist of a small number of packets are not specifically catered for. If such flows form only a small fraction of the total network traffic, then the framework will still work, but router queue

lengths will be nominally higher. If short-duration traffic flows are a substantial fraction of the network's load, then some other strategy will be required to deal with them.

In any event, short-duration flows should be identified with a special **Flow_Id** value (such as 0) within the framework, as a flag to prevent the creation and destruction of ABT entries over a short period of time.

7.8 Bandwidth Reservation and Resource Monitoring

The RBCC algorithm divides router interface capacity fairly amongst traffic flows, while respecting rate limitations set by other routers. If more flows cross through the router, existing flows may have their rate allocations reduced.

The sources of all traffic flows learn about their new rate allocations. For applications which require a *minimum* transmission rate, the new allocation may be unacceptable. As the framework stands, nothing can be done in this situation.

However, it is relatively straight-forward to alter the RBCC algorithm to *prioritise* certain traffic flows: these priority flows will obtain rate allocations first. An example priority scheme would be to have flows with fixed and equal minimum/maximum rate requirements. These would be given priority over other traffic flows. Once the flow's rate is allocated by a router, it cannot be changed while the flow lasts. If there is not enough available bandwidth at a router to sustain a 'fixed rate' flow, its source can be informed and terminate the in-progress connection.

This example priority scheme, in essence, allows 'fixed rate' flows to reserve bandwidth across the network for the flow's duration, while still providing some bandwidth for 'arbitrary rate' flows. The scheme is easily accomplished by ordering the ABT lists in priority order first, and then by ascending limiting rate order. The current Internet architecture does not allow such bandwidth reservation.

Flow priority and bandwidth reservation schemes, while possible in the congestion control framework outlined, are strictly outside the scope of this thesis, and will not be addressed further.

RBCC routers, with Allocated Bandwidth Tables, are in an ideal position to monitor and enforce the sustainable rates set by the distributed RBCC algorithm. A badly-behaved source which exceeds its rate allocation must still have its packets forwarded by the routers who set the rate allocation. They should be allowed to drop some of the source's packets to keep its transmission within the allocated rate.

Again, the monitoring and enforcement of rate allocations is outside the scope of this thesis: I am only considering well-behaved traffic sources. However, as with flow priorities, this area would be a fertile ground for further research.

7.9 Acknowledgment Flow Problems

One severe problem with the congestion control framework, not yet addressed, is the determination of a good desired rate for a destination's acknowledgment flow. Section 6.3.2 gives an equation to determine an instantaneous desired rate for the acknowledgment flow, but in practice this equation will give fluctuating desired rate values.

Consider: The source transmits packets (evenly spaced by Leaky Bucket) to the destination. Packets may be variable in size. Most transport protocols acknowledge packets rather than data bits. Therefore, a constant bit-rate but variable packet-rate data flow will create a variable packet-rate *and* bit-rate acknowledgment flow. If the sizes of incoming data packets are unpredictable, then the destination cannot accurately determine a good desired rate for its acknowledgment flow.

There is no optimum solution for this problem. Data packet sizes cannot be made constant. A transport protocol which acknowledges data bits rather than data packets could be constructed, but it must deal with acknowledgment of retransmitted and out-of-order data and still keep a constant acknowledgment flow rate: this would be difficult to achieve.

Using low-pass or adaptive filters on the instantaneous desired rates calculated by the equation in Section 6.3.2 would produce an average desired rate, but this would not be optimal. In the simulations described in the next few chapters, I chose to use the highest desired rate ever produced by the equation: this wastes bandwidth in the direction of acknowledgment transmission, but prevents oscillations if the instantaneous desired rate values were used. An adaptive filter would be a much better solution.

7.10 Packet Admission and Readmission

One of the requirements for the rate-based congestion control framework is that:

> Packet admission and readmission should be smooth. Bursty traffic (such as is seen during window updates) not only causes short-term congestion but increases end-to-end and round-trip variance. Transport protocols should admit packets into the network as evenly spaced as possible, and routers should attempt to preserve this spacing.

As presented, an RBCC router makes no attempt to preserve packet spacing, despite the requirement above. Simulation results, given in Section 10.10, show that despite this lack of router functionality, inter-packet spacing is well preserved across a network of RBCC routers. Functionality to perform smooth packet readmission by RBCC routers is not required.

7.11 Summary

We have addressed some issues concerning the architecture, implementation and performance of the rate-based congestion control framework. The complexity of the framework should not add an insurmountable load on the network routers, and it removes much of the work that traditional end-to-end transport protocols must perform in terms of flow and congestion control. The framework can also be configured to allow bandwidth reservation or priorities to certain traffic types, and it can monitor and enforce all allocated rates. The overhead of the extra congestion control fields in all packets is significant, but this can be addressed with appropriate hashing techniques.

The identification of a traffic flow was examined. There are several possibilities here, each with their own advantages. After considering all the alternatives, I resolved that a traffic flow within the framework is the flow of packets which conveys data between two applications. With

this option chosen, the intercommunication responsibilities between the Transport Layer and the Network Layer were enumerated.

Several problems have been highlighted in the congestion control framework, notably the problem of determining desired rates for acknowledgment flows. Crude solutions to these problems exist, but the penalty is non-optimum network usage. Further research is needed to address these shortcomings.

In order to demonstrate the effectiveness of the framework in 'real' situations, I now turn to the examination of an implementation of the framework in a packet-based wide-area network simulator.

Chapter 8

Implementing TRUMP/RBCC in the REAL Network Simulator

In order to test the rate-based congestion control framework in a complex environment which might highlight problems or undesirable behaviour, it was decided to simulate the behaviour of the framework. I began initial work on a home-grown network simulator, but this work was discarded when version 4 of the REAL network simulator was obtained.

The REAL network simulator is designed for testing congestion and flow control mechanisms. It was written by S. Keshav and is copyright by the University of California, Berkeley. A description of an early version of REAL is given in [Keshav 88], and REAL was used to perform the simulations described in [Keshav 91]. REAL's author states that "[REAL] has been installed at over 300 sites (but I have no idea how many people actually use it)".

REAL was chosen as the network simulator to test my congestion control framework for the following reasons:

- The simulator was available in full source code form, which allowed me to implement my own congestion framework.
- The simulator was known to run on several hardware platforms.
- The simulator already had several transport flow control schemes and router congestion control schemes already built into it. I would be able to compare my congestion framework against these schemes.

The main alternative to REAL, x-sim[1], was not publically released until after substantial modifications to REAL had been made, in order to implement my congestion framework. The latest version of REAL, version 5, became available after all of my experimental results had been obtained with version 4.

8.1 The REAL Simulator

The REAL 4.0 user's manual describes REAL thus:

[1]Written by Norm Hutchinson and Larry Peterson and others at the University of Arizona and elsewhere.

> REAL is a simulator for studying the dynamic behaviour of flow and congestion control schemes in packet switch data networks. It provides users with a way of specifying such networks and to observe their behavior. Source code is provided so that interested users can modify the simulator to their own purposes.
>
> The simulator takes as input a scenario, which is a description of network topology, protocols, workload and control parameters. It produces as output statistics such as the number of packets sent by each source of data, the queueing delay at each queueing point, the number of dropped and retransmitted packets and other similar information.
>
> There are 11 source types, corresponding to 11 transport protocol and workload types. The sources can be categorized into one of two types: flow-controlled and non-flow-controlled data sources. Flow-controlled sources use acknowledgments and timeouts to implement a reliable transport layer functionality over a lossy network. Non-flow-controlled sources generate data from a known distribution and do not provide a reliable transport functionality; they model cross traffic.
>
> The router implements several scheduling disciplines including:
>
> **fcfs:** First-Come-First-Served.
>
> **fq:** Fair queueing, as described in [Demers *et al* 89].
>
> **fqbit:** This is identical to fq, except that it does bit-setting on packets. It is described in [Demers *et al* 89].
>
> **hrr:** This implements the HRR discipline described in [Kalmanek *et al* 90].
>
> **decbit:** This implements DEC bit-setting described in [Ramakrishnan & Jain 90], and FCFS queueing.

REAL 4.0 also provides several packet dropping mechanisms in routers:

Drop Tail: The newly-arrived packet is dropped.

Drop Head: The oldest queued packet in the router is dropped.

Drop Random: An already-queued packet in the router is randomly chosen and dropped.

Decongest First: The traffic flow with the most bytes queued in the router is found, and the most recently-arrived packet for the flow is dropped.

Decongest Last: The traffic flow with the most bytes queued in the router is found, and the oldest queued packet for the flow is dropped.

8.2 Changes to the Simulator

REAL 4.0 as distributed did not run on any of the platforms available within my department (mainly SPARC machines running Solaris). A slow SPARC machine running SunOS 3.5 was located in another department, and initial simulator use was on this machine.

During the course of my studies, REAL 4.0 was ported to the i386/486 platform under FreeBSD 2.x. This involved some substantial changes to the simulator, mainly to the assembly code which performs thread scheduling. At the same time, I cleaned up the code somewhat in order to reduce the number of warnings generated by the Gnu C compiler.

As well as porting REAL 4.0 to the i386/486 platform, I made further changes and additions to it in order to implement the TRUMP transport protocol and the RBCC rate-measurement algorithm. The changes required were:

- Support for the TRUMP node type and for RBCC_QUENCH and RBCC_TIMER packet types. Files affected: kernel/config.h.
- Extra code to allow an arbitrary number of arguments to be passed to each node. Files affected: kernel/parameters.h, kernel/sim.c, lang/lang.lex and lang/lang.yacc.
- Extra fields in each packet as required by RBCC, and data structures to hold the Allocated Bandwidth Tables. Files affected: kernel/types.h
- New results output code to bypass the limit on the number of open file descriptors per process. Files affected: kernel/plotting.c
- Extra code to deal with the Allocated Bandwidth Tables and detection of traffic flows. Files affected: router/queues.c
- Extra code to perform manual route changes and to plot the number of bytes dropped per router. Files affected: router/router.c and router/router.h
- Extra code to plot the sequence numbers received at the destination. Files affected: sources/sink.c

Three new C files were added to the simulator in order to implement the RBCC algorithm, the TRUMP source and the TRUMP destination: router/rbcc.c, sources/trump.c and sources/trsink.c, respectively.

The simulator was also modified to include an implementation of TCP Vegas, described later in Section 8.5. REAL 4.0 with these changes has been tested on both a SunOS 3.5 machine and a FreeBSD 2.x machine. Instructions on how to obtain the modified REAL simulator are given in Appendix G.

All modifications and extensions to REAL were managed with the RCS source code control system [Tichy 87]. The results in the following chapters were obtained with the following files:

File	Version
sources/jk_reno.c	1.4
sources/ftp_vegas.c	1.11
sources/sink.c	1.3
sources/trump.c	1.23
sources/trsink.c	1.15
router/router.c	1.19
router/rbcc.c	1.53

8.3 Implementation of TRUMP

REAL does not simulate all aspects of the network. For example, although packets have a size depending on their contents (data, acknowledgments etc.), no actual contents are exchanged between the source and the destination. Following this, it makes sense only to implement those parts of a protocol which can be simulated in REAL. The implementation of TRUMP in REAL concentrates mainly on the flow-control aspects of the protocol.

TRUMP is implemented as two source files in the simulator; `sources/trump.c` containing the code for the source, and `sources/trsink.c` containing the code for the destination. The two files contain 400 lines and 140 lines of C code[2], respectively.

8.3.1 TRUMP Destination Code

The TRUMP destination code is relatively trivial, but is complicated by the fact that selective acknowledgments (and their size) may or may not be chosen at run-time for each scenario simulated. The TRUMP destination code has three main modes of operation: setting up a connection, acknowledging data packets, and closing the connection down. See Section B.5 for a proper state diagram of the TRUMP receiver.

A received SETUP packet indicates the start of a connection; a special acknowledgment (SACK) packet is returned in response to this. The destination responds to any SETUP packet received with a SACK packet.

If a DATA packet is received, it must be acknowledged. If selective acknowledgments are *not* being used, an acknowledgment (ACK) packet is returned to the source immediately. If selective acknowledgments are being used, a selective ACK packet is built up, and transmitted if the sequence number of the newly-arrived DATA packet cannot be acknowledged in the ACK's bitmap.

A received connection TEARDOWN packet indicates that the connection is being closed, and is acknowledged with a returned TEARDOWN packet. If there is any outstanding selective ACK packet, this is also returned to the source.

With all returned packets, the source and destination address fields are swapped, the most recent **Rate** field from the source is copied into the **Return_Rate** field. The **Bottle_Node** field is set to identify the destination. The **Desired_Rate** field is set as follows:

$$Desired\_Rate = \frac{Rate * acknowledgment\ packet\ size}{Size\ of\ data\ packets\ acknowledged} \tag{8.1}$$

where **Rate** is the most recent **Rate** value received by the destination. Finally, the **Rate** field in the outgoing packets is initialised to infinity.

8.3.2 TRUMP Source Code

The code for the TRUMP source is more complicated, as it must deal with handshaking and connection termination, rate-based flow control, packet retransmission, Rate Quench packets and handling of parameters provided by the scenario being simulated.

Handshaking and connection termination are performed by a simple state machine as shown in Figure 15 (More accurate state diagrams for the TRUMP protocol are given in Section B.5).

Initially, TRUMP performs a connection handshake by transmitting connection setup packets once a second to the destination until a special acknowledgment (SACK) packet is received. Handshaking can be disabled from the input scenario file as well. In either case, TRUMP moves to the data transmission state, transmitting data packets to the destination. When all have been

[2]Comment lines are excluded, and the source has been formatted with the *indent(1)* program.

acknowledged, TRUMP moves to the connection teardown state; teardown packets are transmitted once a second to the destination until a teardown packet is returned by the destination. At this point, data transmission is complete, and the TRUMP source typically stops. An input scenario parameter can request that TRUMP redo the connection after a certain amount of delay.

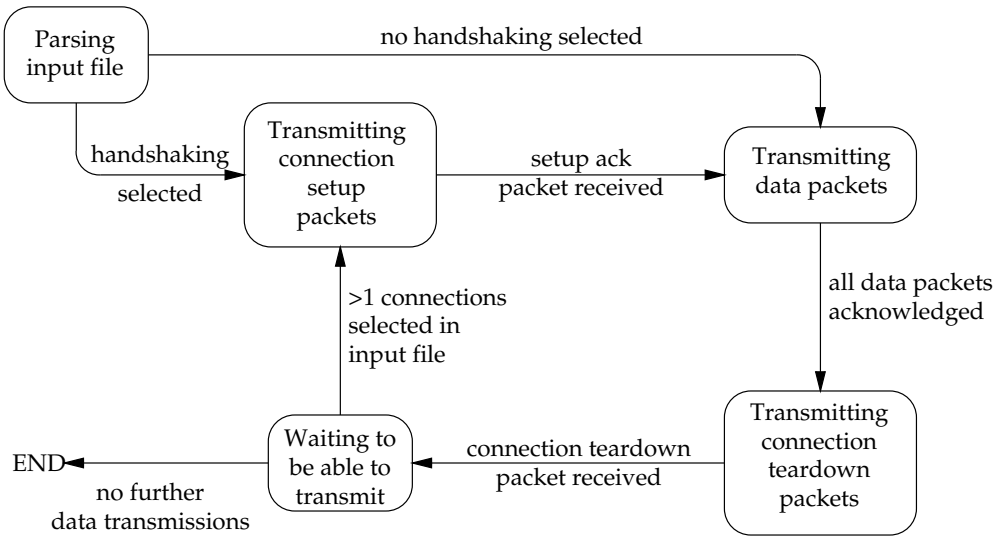


Figure 15: State Diagram for TRUMP Implementation

TRUMP's rate-based flow control is essentially controlled by the **Return_Rate** fields in packets from the destination. The `alter_rate()` function takes the **Return_Rate** fields in ACK and SACK packets and sets the transmission rate accordingly, ensuring that one round-trip's time passes between any rate change. Rate Quench packets lower the transmission rate immediately. Packets are admitted into the network evenly spaced by a delay calculated by Equation 4.1.

There are some cases when TRUMP's rate is not controlled by **Return_Rate** fields. In the connection setup and teardown states, TRUMP transmits acknowledgment-sized packets once a second, and so the transmission rate is known. When handshaking is disabled, TRUMP must begin data transmission with no appropriate rate information; a parameter from the input file sets the initial rate.

The choice of which packet to (re)transmit is performed by `get_seq_no()`. Packets are kept on several queues: the outstanding queue holds those packets which have been transmitted by have not yet been acknowledged; the lost queue holds those packets which have been transmitted and are known to be lost. The queues are ordered by time of transmission. `get_seq_no()` selects the next packet to (re)transmit as follows:

- If any packets are in the lost queue, choose the oldest one (and move it to the newest end of the outstanding queue).
- Otherwise, try to send a new packet. Choose the next unused sequence number, and place a copy of the packet on the newest end of the outstanding queue.
- If there are no new packets left to send, choose the oldest packet on the outstanding queue, and move it to the newest end of the outstanding queue.
- If there no new packets left to send, and nothing in either of the outstanding or lost queues, indicate that data transmission is complete.

Acknowledgment reception is handled by `find_outstanding_pkts()`, which removes

packets from the outstanding queue, and discards them if successfully received, or moves them to the newest end of the lost queue if they were not successfully received.

In order to allow TRUMP's behaviour to be controlled from the input network scenario file, rather than via recompilation, TRUMP accepts many parameters from the input file. Some of these parameters are expressed using the original REAL NetLanguage, and others are expressed using my *params* modification to REAL and are parsed by `parse_opts()`. The parameters accepted by TRUMP, and their influence on TRUMP, are given below.

dest: The number of the TRUMP destination node.

start_time: The time in seconds when TRUMP starts transmitting.

num_pkts: The number of packets to transmit.

peak: The desired transmission rate in bits per second. The default value of 0.0 represents ∞.

noretx: If specified, TRUMP does not retransmit lost segments. The default is to retransmit lost segments.

noshake: If specified, TRUMP does not perform connection setup handshaking, and the **initrate** parameter must specify the initial transmission rate in bits per second. The default is to perform connection setup handshaking.

restart low high: If specified, TRUMP will re-perform the connection and data transmission. TRUMP waits a random delay in the range [low:high] before restarting.

pktsize low high: If specified, packet sizes are spread evenly in the range [low:high].

8.4 Implementation of RBCC

The implementation of RBCC consists of 630 lines of C code in the file `router/rbcc.c`. This file performs the operations of a router, as well as containing the RBCC algorithm.

The file is long and complicated due to the large number of operations it must perform: standard router operations, and RBCC operations. The RBCC/router code performs:

- parsing of the input router parameters,
- identification of new traffic flows,
- flushing of idle or terminated flows,
- deciding to update the Allocated Bandwidth Table (ABT) for packets going from source to destination,
- deciding to update the ABT for packets going from destination to source,
- recalculation of the ABT,
- sending Rate Quench packets,
- queueing/dropping of packets on the output buffer,
- determination of the immediate next route for each packet,
- transmission of packets to the next router or the destination,
- and changing routes.

Router operations will not be discussed in detail, as most of the code was taken directly from the standard REAL router code in `router/router.c`. The only queueing mechanism used in the RBCC implementation is First-Come-First-Served. All of the existing REAL packet dropping mechanisms, enumerated in Section 8.1, are available.

Like TRUMP, RBCC's behaviour can be controlled from the input network scenario file via parameters, and again some parameters are expressed using the original REAL NetLanguage, and others are expressed using my *params* modification to REAL and are parsed by `rbcc_parse_options()`. The parameters accepted by RBCC are given below.

decongestion_mechanism: This selects the packet dropping scheme for when a packet cannot be queued.

policy: The queueing mechanism. Only FCFS can be chosen.

norev: If set, RBCC does not use **Return** packet fields to update the ABTs. The default is to use the **Return** fields.

quench: If set, Rate Quench packets are transmitted as described below. The default is to not send Rate Quench packets.

thresh pkts scale: If set, RBCC will scale the available bandwidth by the given scale if the buffer occupancy is higher than the given number of packets. The default is to do no scaling.

route dest via node at time: If given, the route to the named destination is changed at the given time. A number of route changes can be expressed for each router. The standard REAL router code has been altered to accept this parameter as well.

RBCC checks all setup, setup ack, data and ack packets to see if a new flow is passing through the router. New flows cause the appropriate ABT to be recalculated, in `add_new_conversation()`. Packets which terminate traffic flows cause them to be deleted from the ABTs, and the affected ABTs recalculated.

As route changes may cause 'lost' flows[3] to go undetected, RBCC checks for idle flows every 5 seconds, and flushes them from the ABT with the `flush_lru_conv()` function. This is a very simplistic method to detect lost traffic flows. As each router knows the current data rate of each flow, it should be able to estimate the time when the next packet for the flow will arrive. However, rate changes at the source, packet size changes, short-term congestion, and any associated time variances will render this estimation less accurate. The implementation of RBCC, therefore, errs on the side of simplicity, and checks for idle flows every 5 seconds.

The implementation of RBCC in REAL caches the congestion fields from the last packet seen of each traffic flow. When a new packet arrives, `fields_not_cached()` determines if the cached fields match the new packet's congestion fields; if they do, the ABTs for the traffic flow do not need to be updated.

If the cached fields do not match the packet's fields, RBCC must decide if the ABTs do need to be updated. This is performed by two functions, `do_forward_update_decisions()` and `do_reverse_update_decisions()`. The decision heuristics given in Section 6.2.2 were derived from the implementation's source code. These two functions do not update the ABTs themselves; they only modify the traffic flow information in the ABTs, and indicate that an update is required.

[3] i.e flows no longer crossing through the router.

The RBCC update algorithm, given in Section 6.2.3, is contained in the function `recalc_rate_table()`, and again the algorithm given there was derived from the implementation's source code.

Packets may have their **Rate** and **Bottle_Node** fields updated, reflecting the sustainable rates set in the appropriate ABT. This is performed before the packet is dropped or queued by the `queue_or_drop()` function.

If a data packet causes a flow's rate to be lowered in a router's ABT, a Rate Quench packet is returned to the source with the **Return Rate** field set to the flow's new rate. At present, RBCC routers forward Rate Quench packets without examining their contents, although utilisation of the **Return Rate** field therein may help improve congestion control.

Section 11.3 describes how the use of Rate Quench packets affects the congestion control of the framework.

8.5 Simulation of TCP in REAL

Simulation of my proposed congestion control framework, using TRUMP and RBCC, should indicate its success or otherwise in controlling network congestion. As a reference, each network scenario simulated with TRUMP and RBCC was also simulated using both TCP Reno and TCP Vegas as the transport protocol. This should show if TRUMP and RBCC provide better congestion control than the most commonly used end-to-end window-based congestion control scheme.

As distributed, the REAL 4.0 simulator provides two versions of TCP, TCP Tahoe and TCP Reno, as described in the user's manual:

> The [two TCP] sources implement TCP flow control with the window adjustment and other modifications described by V. Jacobson, M. Karels, P. Karn and C. Partridge [Jacobson 88] [Karn & Partridge 87].
>
> **jk_tahoe:** This implements the flow control in 4.3BSD-Tahoe. The window size can increase or decrease — decreases cause the window to be shut down to 1. Duplicate acks cause retransmission.
>
> **jk_reno:** This incorporates further modifications made to TCP by V. Jacobson (as of Aug '90). When duplicate acks are seen, the window shuts down to half its previous value, not to 1.

As noted in Section 2.3.4, TCP Reno added Fast Recovery, TCP header prediction and delayed acks to TCP. The REAL version also faithfully implements the coarse timers from the 4.3BSD-Reno operating system implementation of TCP. TCP Reno was chosen over TCP Tahoe as a reference congestion control scheme for simulations, as the modification by Jacobson give Reno much better congestion control than Tahoe.

Since REAL 4.0 was made available, further enhancements to TCP have been proposed. Known as TCP Vegas, they are described in [Brakmo *et al* 94] and [Brakmo & Peterson 95], with several followup papers. The author of REAL made available an implementation of TCP Vegas for the then-unreleased REAL version 5; this implementation was written by Omar Hellal, and its use is described in [Ait-Hellal & Altman 96].

With help from Yoshifumi Nishida, Hellal's implementation was 'back-ported' to REAL 4.0. The port of Hellal's TCP Vegas code to REAL 4.0 has been verified by Hellal and by comparison with the results given in [Brakmo *et al* 94] and [Brakmo & Peterson 95].

TCP Vegas was also chosen as a reference congestion control scheme for simulations, as some of the enhancements in Vegas are claimed to improve TCP's congestion control.

8.6 Measurements Obtained by The REAL Simulator

In the next chapter, I will discuss the results obtained by simulating my proposed rate-based congestion control framework on the REAL network simulator. Firstly, I wish to outline the types of results produced by the simulator, and their relevance to the detection and control of network congestion.

The types of results produced by REAL are as follows:

- Packet sequence numbers and their transmission time at the source.
- Packet sequence numbers and their reception time at the destination.
- Packets dropped and their drop time at each router.
- Windows size for TCP sources, and the number of outstanding packets for TRUMP.
- The end-to-end delay between source and destination.
- The round-trip time between source and destination.
- The actual link utilisation for all links.
- The sustainable rate measured by RBCC as seen at the destination and at the source.
- The number of packets queued on each router's output interface.
- The RBCC sum allocated bandwidth for each router's output interface.

8.6.1 Indicators of Network Congestion

There are several indicators of network congestion:

- Packet loss as seen by a source,
- Increasing end-to-end delays and round-trip times as seen by destinations and sources respectively,
- High buffer occupancy in routers,
- Packet loss in routers, and
- A higher offered load to a link than its bandwidth capacity.

The first two indicators cannot be used by themselves as congestion indicators: packet loss is caused by both network congestion and bit errors; increasing round-trip time is caused by network congestion and lower available bandwidth to the source (competing sources). Window-based transport protocols such as TCP, however, do use these as congestion indicators. The last three indicators are reliable congestion indicators.

One of the best indicators of network congestion is a router's average queue lengths for its output interfaces. If a queue has an average length (in packets) less than 1, that interface is underloaded. If a queue has an average length greater than 1, then it is receiving more packets than it can transmit, and is overloaded. A queue length of 1 indicates an optimally loaded interface. As a queue grows in length, packets suffer delays in retransmission at the router, increasing end-to-end delays and round-trip time. All queues are finite in length, and a router must drop packets when there is no space left to queue incoming packets.

8.6.2 Other Indicators of Network Performance

Although not strictly related to network congestion, these factors show how well a transport protocol is using the resources on the network.

Packet transmission times can be used to infer instantaneous and average data rates. A source which successfully completes its transmission before an equivalent source using a different transport protocol has made better use of network resources.

End-to-end and round-trip times with high variance cause problems for protocols which use round-trip time prediction. Smooth end-to-end times are important for time-sensitive data transmission such as voice and video.

8.7 Summary

In order to measure the congestion control performance of the proposed rated-based framework, it has been implemented in the REAL network simulator. This simulator was designed to study flow and congestion control in connectionless packet-switched networks, and provides several end-to-end flow & congestion control mechanisms, as well as several router-level packet queueing and packet dropping mechanisms.

The two new functional elements of the rate-based framework, the TRUMP Transport protocol and the RBCC Sustainable Rate Measurement function, have been implemented within the REAL simulator. As well, the proposed modifications to TCP's congestion control, known as TCP Vegas, have also been implemented within the REAL simulator. The next three chapters examine the congestion control performance of the the rate-based framework, using TRUMP and RBCC, for several network scenarios. The congestion control performance of two end-to-end based systems, TCP Reno and TCP Vegas, will also be examined, and compared with the framework's performance.

Chapter 9

Results of Simulating TRUMP and RBCC in REAL

9.1 Introduction

Testing of the rate-based congestion control framework was performed on a number of hand-crafted network scenarios, as well as a large number of pseudo-randomly generated scenarios. The hand-crafted scenarios are described here.

The purpose of testing TRUMP/RBCC on hand-crafted scenarios is to demonstrate its behaviour where the points and severity of congestion can be determined in advance. As well, specific characteristics of the framework (such as fairness, ability to deal with route changes, ability to work with desired rates from sources) can be checked.

9.2 Description of Scenarios

Nine scenarios with different characteristics were crafted:

Scenario 1: A simple scenario with one traffic flow, one route and one congestion point.

Scenario 2: A scenario with several traffic flows and one congestion point. Flows start at different times, and the optimal flow rates at any time can be calculated.

Scenario 3: Several flows compete for a high-latency low-bandwidth link. This scenario should reveal any congestion problems caused by delay.

Scenario 4: Several flows come on-line at different times to compete for a low-bandwidth link. This scenario should reveal any temporal unfairness in rate allocation.

Scenario 5: Several flows with different route lengths compete for a low-bandwidth link. In this scenario, there are many cross-traffic flows with very short route lengths. This scenario should also reveal any spatial unfairness in rate allocation.

Scenario 6: A single long-haul traffic flow competes against a number of short-haul cross-traffic flows. Again, this scenario should also reveal any spatial unfairness in rate allocation.

Scenario 7: A scenario similar to Scenario 2, with packet sizes evenly distributed across a range. This should show any problems with varying packet sizes in TRUMP/RBCC.

Scenario 8: A scenario similar to Scenario 2, but with a route change during the scenario. This scenario should show if, and how well, TRUMP/RBCC can adapt to a route change in traffic flows.

Scenario 9: A modification to Scenario 6 to examine the effect of Rate Quenches on large selective acknowledgment delays.

9.2.1 Scenario Variations

There are three variations of each scenario:

TRUMP/RBCC All sources use TRUMP and all destinations use the TRUMP sink code. All routers use RBCC.

TCP Reno All sources use TCP Reno and all destinations use the standard REAL sink code. All routers are standard REAL routers.

TCP Vegas All sources use TCP Vegas and all destinations use the standard REAL sink code. All routers are standard REAL routers.

Nearly all scenarios use the same global parameters (the `real_params` sections shown in Appendix E). The most important parameters are given below:

- Acknowledgment packets are 40 octets in length, the same as TCP/IP acknowledgment packets in the Internet.
- Data packets are 1500 octets in length.
- The buffer size for the output interfaces in each router is 15,000 octets in length.
- For TCP Reno and TCP Vegas, the maximum window size is 600 segments. No source in the scenarios simulated reaches this value.
- TRUMP performs selective acknowledgments, with 1 data packet acknowledged in every acknowledgment packet. As will be seen in Chapter 10, >1 packets acknowledged per ack packet causes higher end-to-end and round-trip variance.
- RBCC scales an output interface's bandwidth by 0.75 when the output buffer for that interface reaches 5 packets or more. This congestion avoidance heuristic is described in Section 6.3.6.
- Routers use First-Come-First-Served as the packet selection method and Drop Tail (i.e the last packet to arrive) as the packet dropping method.
- Utilisation for each linked is averaged over a period of 2 seconds. The data obtained by REAL for link utilisation is coarse. REAL periodically calculates the link utilisation by dividing the amount of data transmitted over the link by the time period. If the period is too small, the utilisation information is inaccurate, being sensitive to packet scheduling etc. If the period is too long, the utilisation value is more accurate, but is the average utilisation over a larger time period.

Scenarios which do not use the parameters given above will be noted. The REAL input files containing these scenarios are given in Appendix E.

9.2.2 Tabled Results

In this chapter, a per-source table of results will be given for each scenario. The columns in the table hold:

- The end time of the source's data transmission, in seconds.
- The average data transmission rate, in bits per second. This includes the header in each packet, and not just the data payload.
- The average end-to-end time in seconds, and the standard deviation of the end-to-end time.
- The average round-trip time in seconds, and the standard deviation of the round-trip time.
- The number of data packets lost, and the number of data packets that were retransmitted.

9.3 Scenario 1

Scenario 1 is shown in Figure 16:

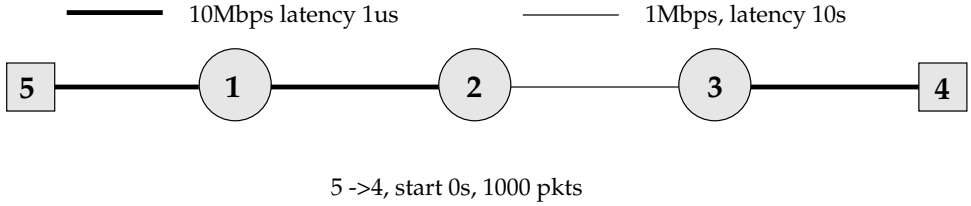


Figure 16: Scenario 1

This simple scenario is designed to show the transmission characteristics of the different transport protocols when there are no conflicting sources. The low-speed link $2 \rightarrow 3$ will limit the data rate of the data source at node 5. Router 2 may need to buffer packets if the source of traffic tries to exceed the available bandwidth.

Throughout this chapter, diagrams of network scenarios show squares as hosts, circles as routers, thick lines as high-speed links and thin lines as low-speed links. The speed and latency of each link is also given, as is the start time and number of data packets in each traffic flow.

9.3.1 Transmission Sequence Numbers

With all source types, node 5 begins transmission at 0 seconds. With TRUMP, the return handshake packet from node 4 reaches 5 at time 0.0109 seconds with a **Return Rate** value of exactly 1Mbps. Node 5 transmits data packets at this rate until time 12.0109, as shown in Figure 17.

TCP Reno does not perform a handshake, and begins data transmission at 0 seconds[1]. Acknowledgment packets open the window size (and hence the transmission rate) until a packet is lost by router 2 at time 0.154. The negative acknowledgment reaches the source at time 0.319, and it lowers its window size from 23 packets back to 14. Reno transmits data packets until time 12.912.

[1]In fact, for all scenarios, TCP Reno and TCP Vegas get the jump on TRUMP by transmitting without performing any handshaking. Similarly, neither Reno nor Vegas perform connection termination.

TCP Vegas also does not perform a handshake, and begins data transmission at 0 seconds. Its data transmission rate curve[2] is extremely similar to TRUMP's, except for some small deviations before time 0.3. Data transmission continues until time 12.018.

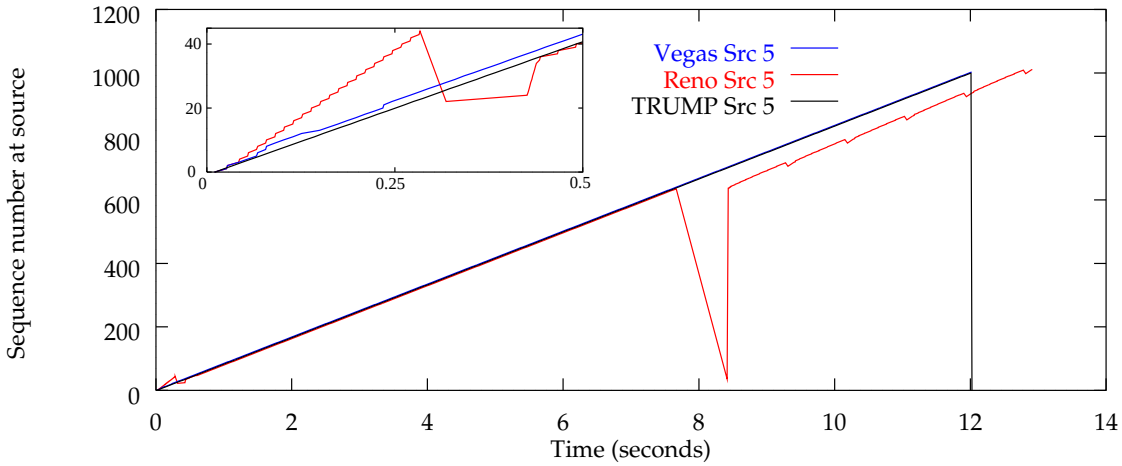


Figure 17: Sequence Numbers in Scenario 1

9.3.2 Router Queue Lengths

For TRUMP, buffer queue lengths in routers 2 and 3 never exceed 1 packet. No packets are dropped by any router. With Reno, buffer queue lengths in router 2 reaches the maximum value of 10 data packets several times as shown in Figure 18. Overall packet loss is 18 packets, all dropped by router 2. With Vegas, no packets are lost by any router, and the buffer queue length in router 2 sits at between 2 and 3 data packets for nearly all of the transmission time.

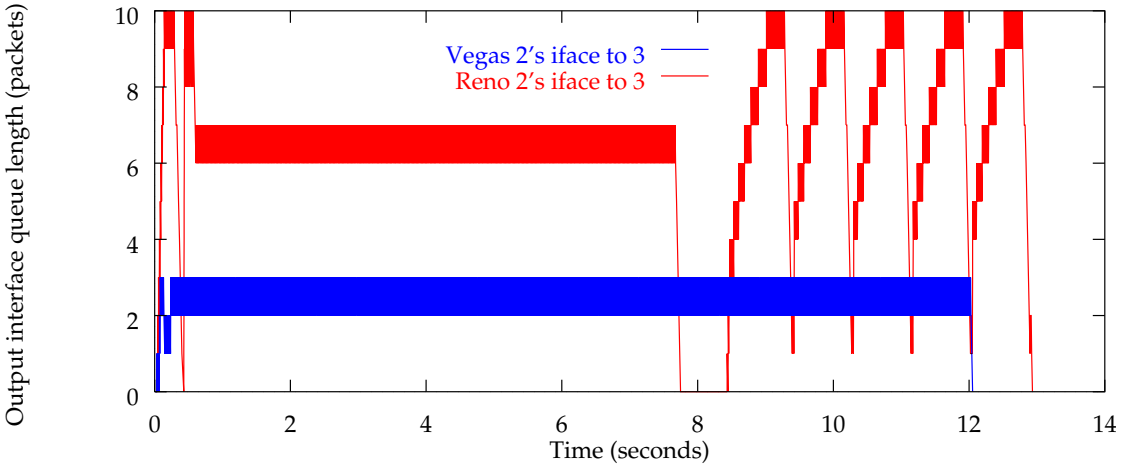


Figure 18: Queue Lengths in Scenario 1

[2]Strictly speaking, the derivative of the sequence number curve.

9.3.3 Round-Trip Times

The round-trip time for all TRUMP acknowledgments is 0.016 seconds, as shown in Figure 19. The round-trip time for Reno oscillates between 0.083 and 0.126 seconds, reflecting the queue size in router 2. In Vegas, the round-trip time rises quickly from 0.016 seconds to 0.047 at time 0.6, and maintains a constant value of 0.048 for most of the transmission time.

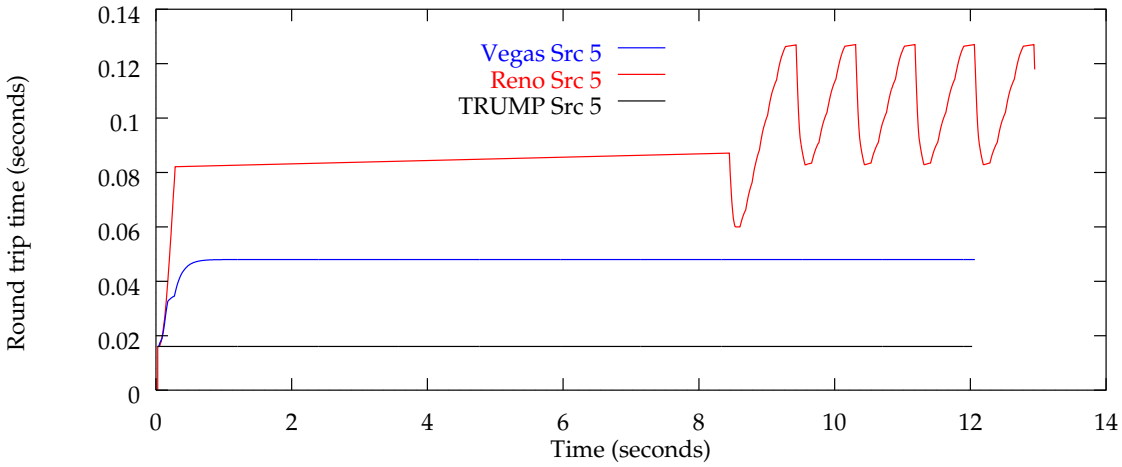


Figure 19: Round Trip Times in Scenario 1

In summary, TRUMP and Vegas do not lose packets in Scenario 1, but Reno does. TRUMP's and Vegas' data rates are extremely similar; Vegas, however, maintains a higher queue size in router 2, and this is reflected in the higher round-trip time.

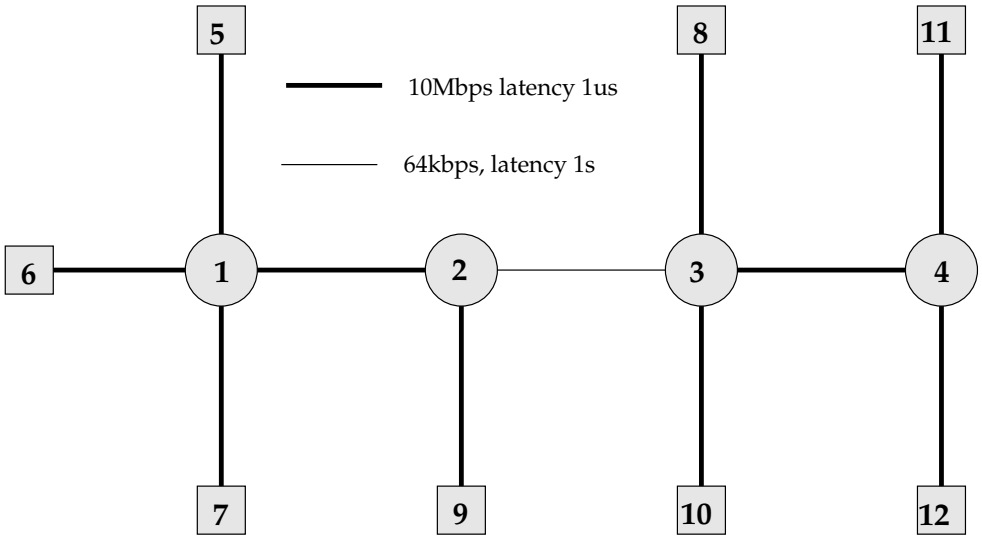
9.4 Scenario 2

Scenario 2 is shown in Figure 20:

The scenario shows a number of nodes connected by full-duplex links. All links have a data rate of 10Mbps, except for the link between routers 2 and 3, which is a 64kbps link. The 10Mbps links have a latency of 1 μsec, and the 64kbps link has a latency of 1 second. The latter latency, although unrealistic, serves to set the overall round-trip time of this small network to that which is often experienced in a global-scale network.

There are four data flows, $5 \rightarrow 10$, $6 \rightarrow 11$, $7 \rightarrow 12$ and $8 \rightarrow 9$. All cross the 64kbps link, but all start at different times. Thus the link between routers 2 and 3 is the bottleneck in the scenario, and should cause congestion in those routers.

Note that all data flows have corresponding acknowledgment flows in the reverse direction. Note also that the data flow $8 \rightarrow 9$ is in the reverse direction to the other data flows.



5 ->10, start 0s, 1000 pkts

7 ->12, start 12s, 1000 pkts

6 ->11, start 140s, 100 pkts

8 -> 9, start 18s, 1000 pkts

Figure 20: Scenario 2

9.4.1 Optimum Rates for Scenario 2

Given Scenario 2, it is simple to compute the optimum rates for each of the sources; the calculations are given in Appendix F. The following table summarises the rates; the parameter t is the simulation time in seconds. The value Off_x indicates the time that source x stops transmitting data. These optimum rates provide a baseline to compare the rates calculated by the RBCC scheme.

Time t (seconds)	**Optimum Rate (bps)**			
	Source 5	Source 7	Source 8	Source 6
$0 <= t < 12$	64,000			
$12 <= t < 18$	32,000	32,000		
$18 <= t <= 140$	31,168.8	31,168.8	62,337.7	
$140 <= t <= Off_6$	20,779.2	20,779.2	62,337.7	20,779.2
$Off_6 <= t <= Off_8$	31,168.8	31,168.8	62,337.7	
$Off_8 <= t$	32,000	32,000		

9.4.2 Transmission Sequence Numbers

Packets are transmitted by a source in monotonically increasing sequence number, barring retransmissions, with the derivative of sequence number versus time giving the effective transmission rate of a source. Figure 21 show the sequence number for each packet transmitted by sources 5 and 6 using TCP Reno, TCP Vegas and TRUMP.

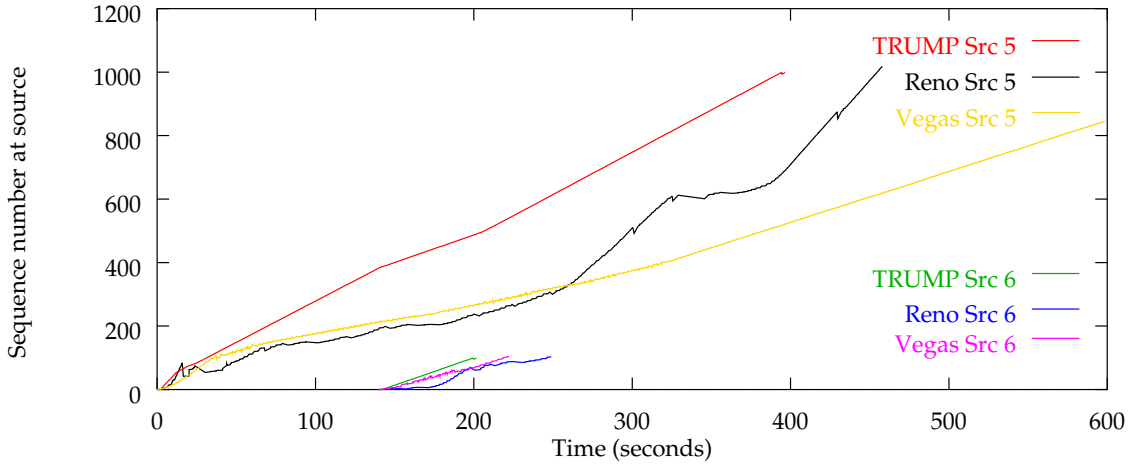


Figure 21: Sequence Numbers for Sources 5 and 6

TRUMP's sequence numbers rise smoothly, controlled by the rate values calculated by RBCC. Reno's rate is much more jagged, caused by its window size changes which reflect packets lost by the destination. Note in particular the early rate fluctuation for Reno's source 5. Vegas' rate is much smoother than Reno's, and its startup is much better behaved than Reno's: however, Vegas' source 5 does not detect that source 6 has terminated at time 222, and so its throughput is even worse than Reno source 5.

The plot for sources 7 and 8, Figure 22, shows similar characteristics. Both TCP Reno and Vegas transmits packets for source 8 (the reverse data flow) at a much slower rate than TRUMP/RBCC. This is surprising, given the fact that in this scenario TRUMP/RBCC did not lose any packets whatsoever: there must have been adequate bandwidth for source 8 which was not used by both Reno and Vegas.

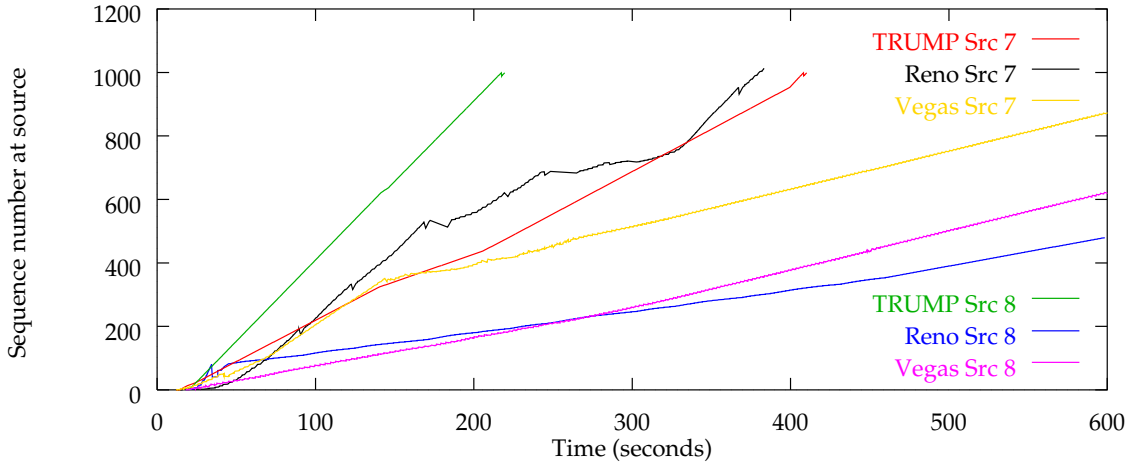


Figure 22: Sequence Numbers for Sources 7 and 8

9.4.3 Packet Loss

In Scenario 2, TRUMP sources do not lose any packets whatsoever. Reno's sources, on the other hand, continually lose packets throughout the life of their transmission. Most packet losses are at the beginning of the data flow (until the optimum window size is reached), and Reno slowly loses packets after that as it tries to open its sliding window to take advantage of any new excess bandwidth. Reno loses 108 packets in total. Figure 23 shows the cumulative packet losses by

TRUMP, Vegas and Reno.

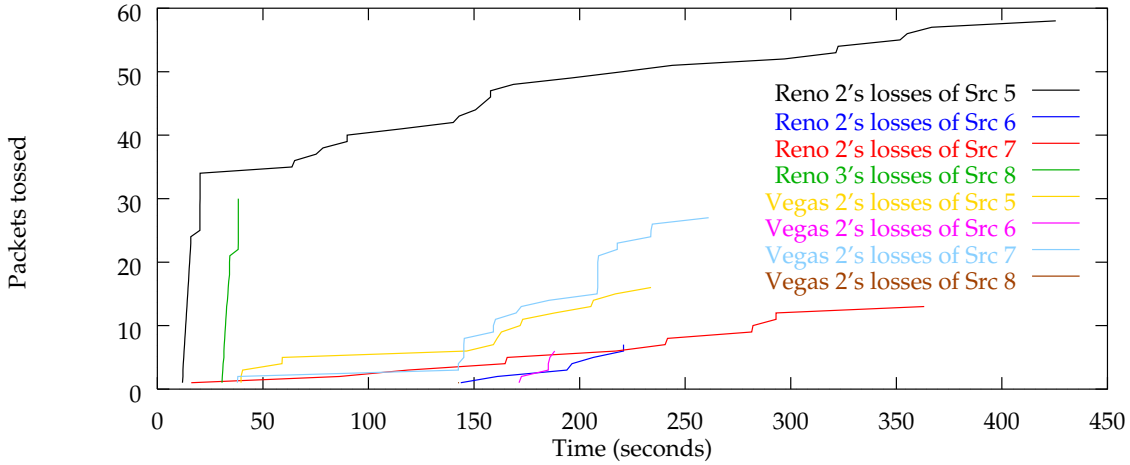


Figure 23: Cumulative Packet Losses by TCP

The Vegas improvements to TCP help reduce packet loss, with sources 5, 6 and 7 losing 16, 6 and 27 packets, respectively. This improvement seems more a result of the failure to utilise bandwidth after source 6 terminates, as against any real improvement to TCP.

9.4.4 Router Queue Lengths

In Scenario 2, routers 2 and 3 can only buffer 15,000 bytes (10 data packets). If the router cannot buffer the packet, it is immediately discarded (FCFS queueing and Drop Tail packet dropping are in use). Figure 24 shows the queue lengths (in packets) for router 2.

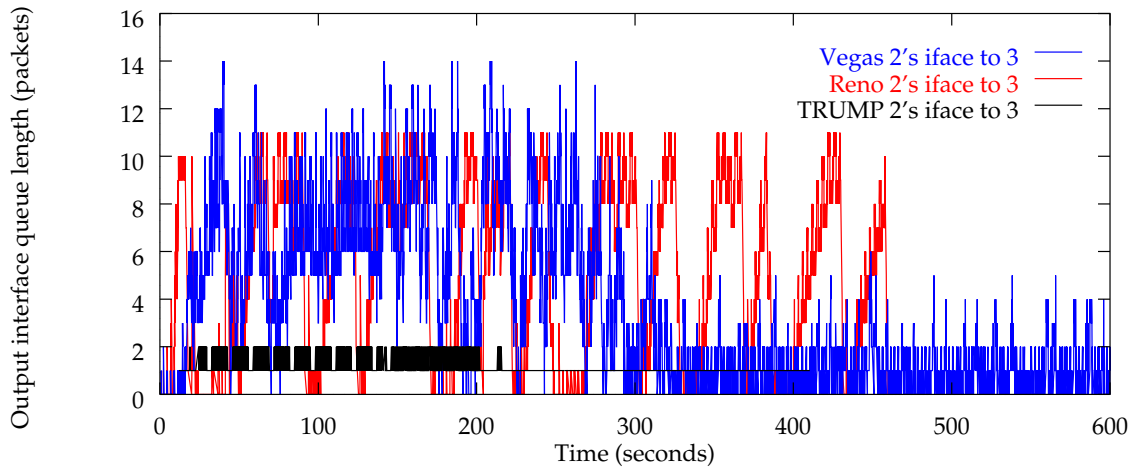


Figure 24: Average Queue Length in Router 2

This plot is cluttered. For router 2, there are several peaks in queue length where Reno's excessive rate has caused lost packets, due to its window resizing mechanism. TRUMP/RBCC keeps the queue length for router 2 between 1 and 2 packets over the entire simulation. Ve-

gas keeps average queue lengths below Reno's when source 6 is idle (due to poor bandwidth utilisation), but causes many peaks above queued packets when source 6 is transmitting.

9.4.5 End-to-End Delays

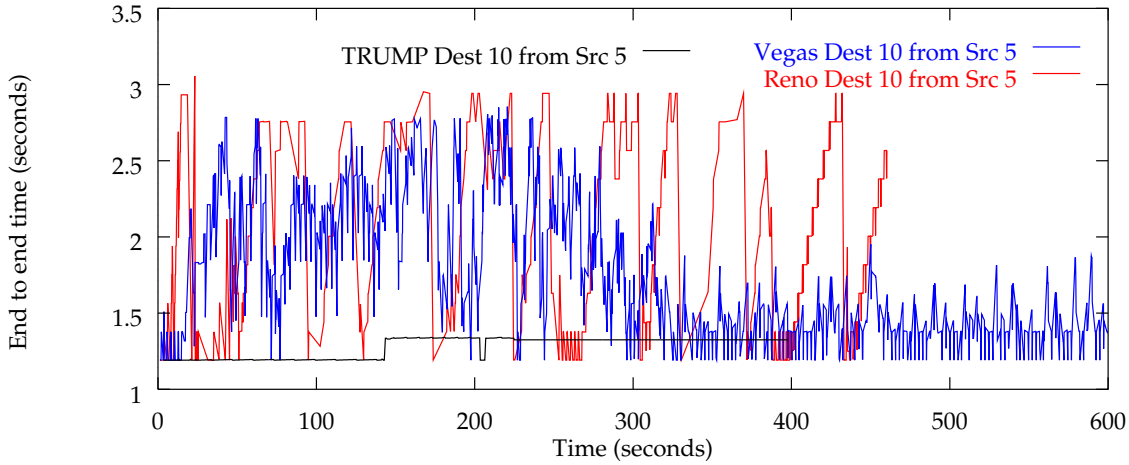


Figure 25: Source/Destination Delay between 5 and 10

In Scenario 2, the end-to-end delay for a packet from source 5 must be at least 1.188 seconds, given the link latencies and packet propagation time. Figure 25 shows the measured end-to-end delay of source 5 for TCP Reno, TCP Vegas and TRUMP. TRUMP has an end-to-end delay between 1.19 and 1.34 seconds; Reno's value oscillates between 1.19 and 3.06 seconds, due to the oscillation of the queue size in router 2; Vegas shows faster oscillations which have a higher deviation than Reno's, although the average end-to-end delay is lower. The end-to-end delays for sources 6 and 7 show the same behaviour. The low end-to-end delay deviation exhibited by TRUMP/RBCC is particularly suited for time-sensitive data such as real-time voice or video transmission.

A summary of per-source results for TRUMP, Vegas and Reno is given below.

Source Node	End Time	Avg Rate	End to end Avg	End to end S.Dev	Round-trip Avg	Round-trip S.Dev	Pkts Lost	Pkts Retx'd
TRUMP 5	396.6	30597	1.28	0.07	2.32	0.07	0	6
Vegas 5	598.4	23843	1.77	0.45	2.01	0.74	16	342
Reno 5	457.9	29037	2.06	0.59	3.00	0.47	58	89
TRUMP 6	201.6	20989	1.25	0.00	2.35	0.06	0	4
Vegas 6	222.3	24641	2.30	0.41	2.27	0.95	6	63
Reno 6	248.8	12905	2.01	0.65	2.98	0.52	7	12
TRUMP 7	410.2	30633	1.21	0.02	2.25	0.07	0	11
Vegas 7	599.9	29477	1.72	0.48	1.80	1.19	27	569
Reno 7	383.3	33322	2.19	0.58	3.15	0.47	13	17
TRUMP 8	219.5	60841	1.19	0.00	2.33	0.07	0	11
Vegas 8	599.8	27248	1.37	0.18	0.80	0.21	0	698
Reno 8	598.4	10628	1.38	0.41	2.63	0.43	30	33

Note that end-to-end and round-trip deviation for TRUMP is generally lower then Reno's

and Vegas'. TRUMP's average data rate is higher than Reno's and Vegas', and although Vegas loses fewer packets than Reno, it retransmits substantially more packets than TRUMP or Reno.

9.4.6 Link Utilisations

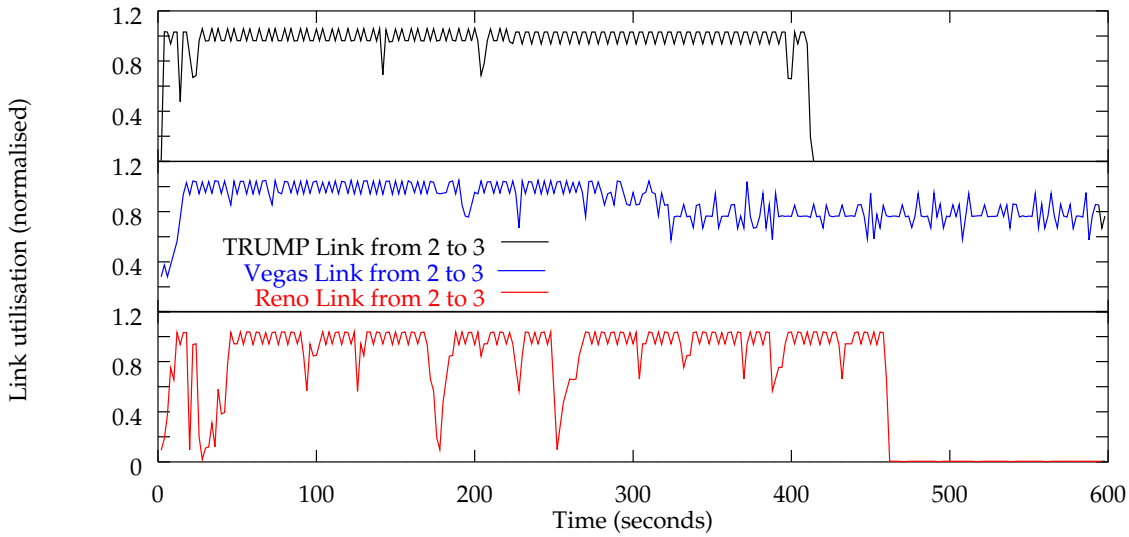


Figure 26: Link 2 → 3 Utilisation in Scenario 2

Another indication of network congestion, closely tied to average queue lengths, is the utilisation of a link. Together with the queue lengths, the link utilisation shows whether an output link is underloaded or overloaded.

Figure 26 shows the utilisation of the link in the direction of router 2 to router 3 for TCP Reno, TCP Vegas and TRUMP/RBCC. TRUMP fully utilises the link, with occasional dips due to rate recalculations as sources start and stop. Vegas performs nearly as well, but with utilisation falling off after source 6 terminates. Reno shows many dips due to window shutdown. The average utilisation for TRUMP, Vegas and Reno is 0.97, 0.87 and 0.83, respectively.

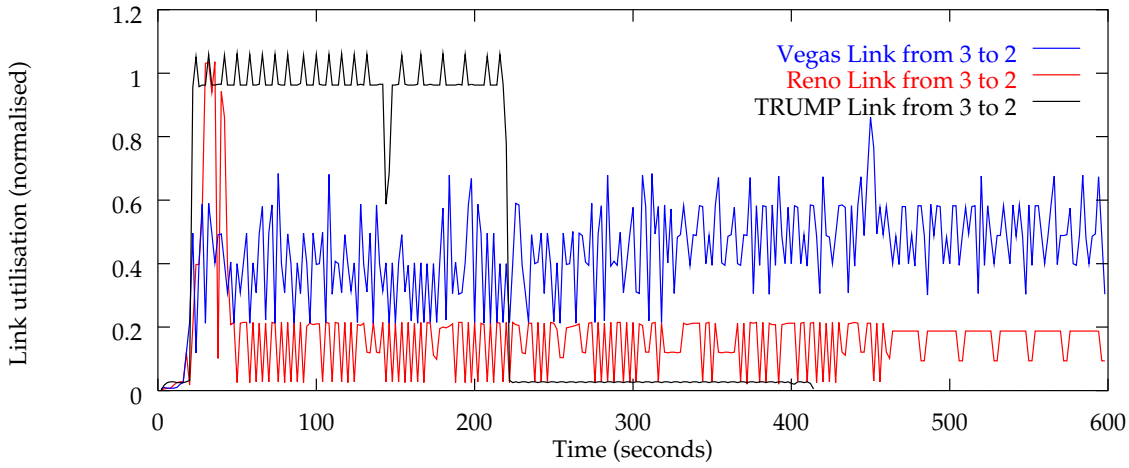


Figure 27: Link 3 → 2 Utilisation

The utilisation of the link in the direction of router 3 to router 2 shows that TRUMP fully utilises the link while source 8 is transmitting, with only acknowledgment traffic passing over the link when source 8 stops. Both Vegas and Reno, on the other hand, never fully utilise this link with traffic from source 8, although Vegas makes around twice the utilisation of the link

compared to Reno.

9.4.7 Predicted and Measured TRUMP Transmission Rates

The optimum rates for TRUMP sources in Scenario 2 were given in Section 9.4.1. Figure 28 shows the actual rates returned from RBCC.

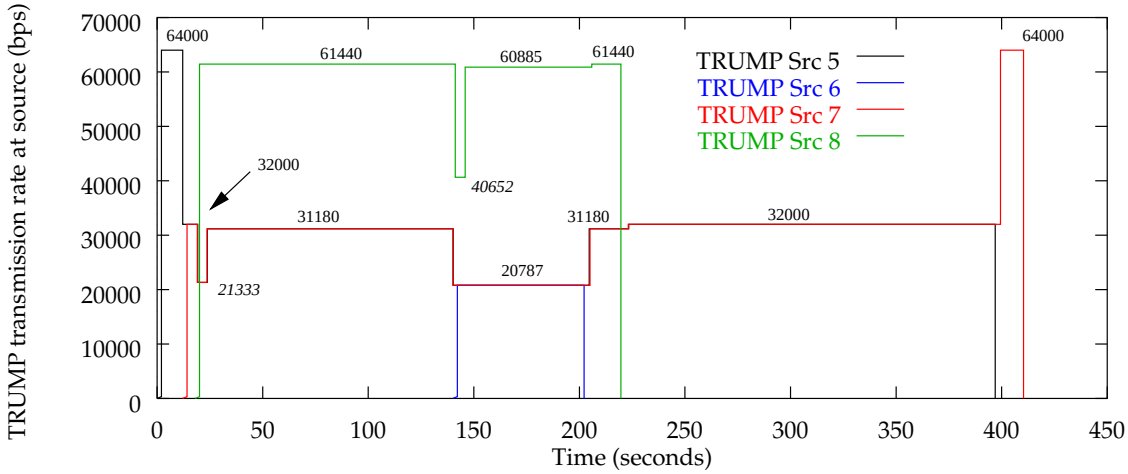


Figure 28: TRUMP's Rates for Scenario 2

As can be seen, the rates used by the TRUMP sources match nearly exactly with the optimal rates. There are slight delays due to round-trips which cause the TRUMP sources to adopt new rates after the optimal time. Initial bit rates are correctly found after TRUMP's two-way connection handshake. The close correspondence between the optimum rates and RBCC's calculated rates indicates that RBCC's rate calculation method is successful, and fair to all traffic sources.

9.4.8 Evaluation of Comparison Results

The comparison of TCP Reno, TCP Vegas and TRUMP/RBCC in Scenario 2 shows that TRUMP/RBCC gives quite different traffic characteristics to both TCP versions, with much less network congestion. RBCC determines source rates which will fully utilise output interfaces (where possible), while maintaining very low output queue sizes in routers. This has the effect of preventing long-term congestion, minimising buffering delays and gives end-to-end delays a very low variance, which is ideal for time-sensitive traffic. The rates calculated by RBCC are quite close to the optimum rates for Scenario 2.

9.5 Scenario 2a

Scenario 2 was modified to limit the desired transmission rates of flow 6 $\rightarrow$ 11 to 18,000 bps, and flow 7 $\rightarrow$ 12 to 32,000 bps. As with Scenario 2, the optimum transmission rates can be calculated:

Time t (seconds)	Optimum Rate (bps)			
	Source 5	Source 7	Source 8	Source 6
$0 <= t < 12$	64,000			
$12 <= t < 18$	32,000	32,000		
$18 <= t <= 140$	31,168.8	31,168.8	62,337.7	
$140 <= t <= Off_6$	22,168.8	22,168.8	62,337.7	18,000
$Off_6 <= t <= Off_8$	31,168.8	31,168.8	62,337.7	
$Off_8 <= t$	32,000	32,000		

Although flow $7 \rightarrow 12$ is bandwidth limited and flow $5 \rightarrow 10$ is not, they should be treated equally when $7 \rightarrow 12$ is allocated a rate of 32,000 bps or less.

The rates calculated by the distributed RBCC algorithms are:

Time t (seconds)	ABT Allocated Rate (bps)			
	Source 5	Source 7	Source 8	Source 6
2.02	64000.0			
14.15	32000.0	32000.0		
22.94	31180.0	31180.0	61440.0	
148.3	22180.0	22180.0	60960.0	18000.0
215.2	32000.0	32000.0	61440.0	

Again, the rates set by RBCC are very close to the optimum rates. As well, the bandwidth limited flows do not have their desired rates exceeded, and they are treated equally to the other flows when bandwidth must be shared.

9.6 Scenario 3

Scenario 3 is shown in Figure 29:

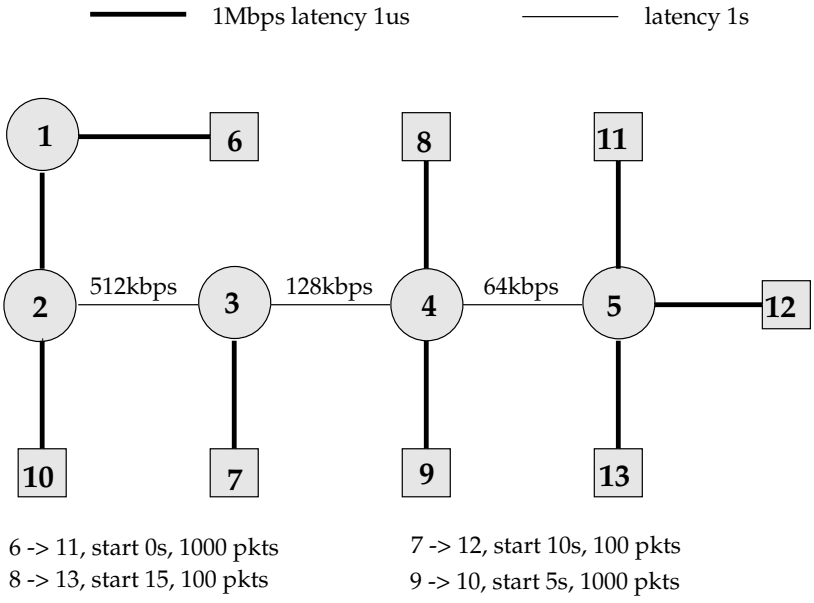


Figure 29: Scenario 3

This scenario is designed to see how well existing traffic sources react to other sources starting and stopping. The three low-speed links have very long latencies, and different capacities from each other. Three traffic flows are from left to right, and the $9 \rightarrow 10$ flow is from right to left. The start times for successive sources is close to the longest round-trip time.

9.6.1 Transmission Sequence Numbers

In Scenario 3, the TCP Reno sources 6 and 9 exhibit an exponentially increasing sequence number curve (and hence data rate curve). Both TRUMP's and Vegas' sequence number curves are linear. Note that Vegas' source 6 has not finished transmitting at time 2000 seconds. The per-source summary table below gives the end times and average data rates for all sources.

Source Node	End Time	Avg Rate	End to end Avg	End to end S.Dev	Round-trip Avg	Round-trip S.Dev	Pkts Lost	Pkts Retx'd
TRUMP 6	244.1	52077	3.41	0.31	6.44	0.33	0	33
Vegas 6	>2000	11919	3.41	0.14	2.10	0.27	0	1055
Reno 6	361.6	34811	4.07	0.56	7.04	0.51	3	5
TRUMP 7	72.7	22359	3.18	0.23	5.23	0.23	0	9
Vegas 7	179.1	14762	2.58	0.36	1.80	0.56	0	106
Reno 7	141.3	10324	3.25	0.47	5.15	0.49	4	6
TRUMP 8	75.5	21538	2.24	0.15	3.24	0.15	0	5
Vegas 8	53.5	35498	1.82	0.42	2.70	0.51	0	3
Reno 8	243.5	35558	1.98	0.49	2.60	0.39	21	27
TRUMP 9	112.9	120597	2.12	0.00	4.15	0.04	0	43
Vegas 9	1403.7	18351	2.17	0.05	1.32	0.28	0	1137
Reno 9	192.7	66890	2.15	0.05	4.16	0.10	0	1

TRUMP/RBCC and Vegas lose no packets, whereas Reno does. Note, however, that Vegas exhibits a large number of retransmissions: this scenario seems to reveal a defect in Vegas' retransmission timer strategy. End-to-end and round-trip deviation is generally lower for TRUMP/RBCC than for either TCP version. TRUMP sources have higher transmission rates than their equivalent TCP sources.

9.6.2 Router Queue Lengths

With TRUMP sources, no routers are congested[3] except for router 4, as shown in Figure 30. Its total buffer queue size for the $4 \rightarrow 5$ link is above 2 packets for time 17 to 76 seconds, spending most of the time at 6 packets queued.

[3] i.e. buffer sizes do not exceed one packet.

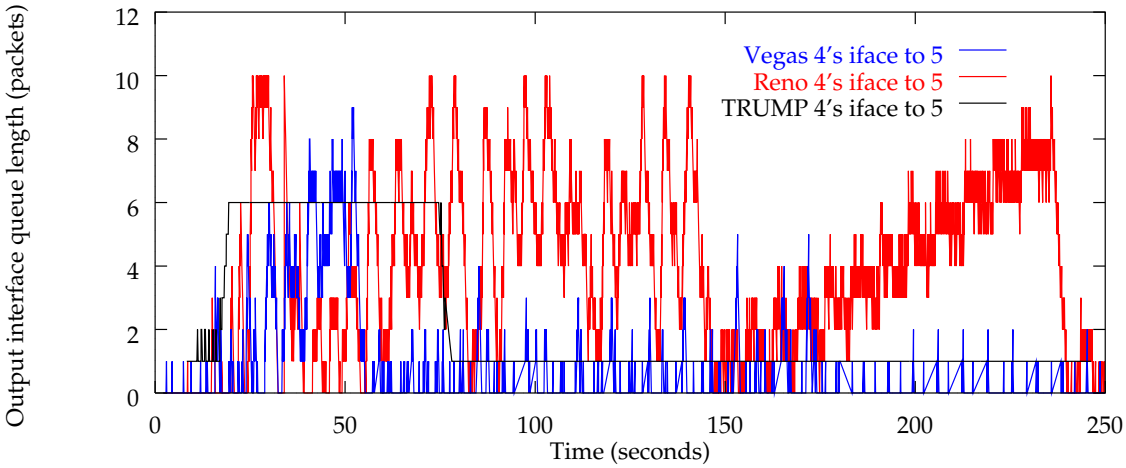


Figure 30: Queue Lengths in Scenario 3

With TCP Reno sources, the following routers are congested:

- Router 2 on the link to 3, with a peak of 2 packets at time 53.
- Router 3 on the link to 4, with 5 peaks to 2 packets, 3 peaks to 3 packets and one peak of 4 packets from time 40 to 200.
- Router 4 on the link to 3, with dozens of peaks above 1 from time 40 to 200.
- Router 4 on the link to 5. This interface is severely congested, with 2 or more packets queued for most of the time 60 to 200, as shown in Figure 30. There are 9 peaks of 10 packets queued.

As with TCP Reno, the following routers are congested with TCP Vegas:

- Router 2 on the link to 3, with several short peaks of 3 packets queued before time 200.
- Router 3 on the link to 4, with many peaks of 2 packets and above, and two short peaks of 4 packets queued before time 200.
- Router 4 on the link to 3, with some peaks of 2 packets and above, and one short peak of 5 packets queued at time 119.
- Router 4 on the link to 3, with dozens of peaks above 1 from time 40 to 200.
- Router 4 on the link to 5. This interface is severely congested from times 35 to 55, with a queue length of 1 to 9 packets, as shown in Figure 30.

9.6.3 Link Utilisations

Utilisation of the links 4 to 5 and 4 to 3 are much better for TRUMP than for TCP, as shown in Figures 31 and 32. In fact, neither of the TCP versions make full use of the link from 4 to 3, and TCP Vegas has much poorer utilisation than TCP Reno.

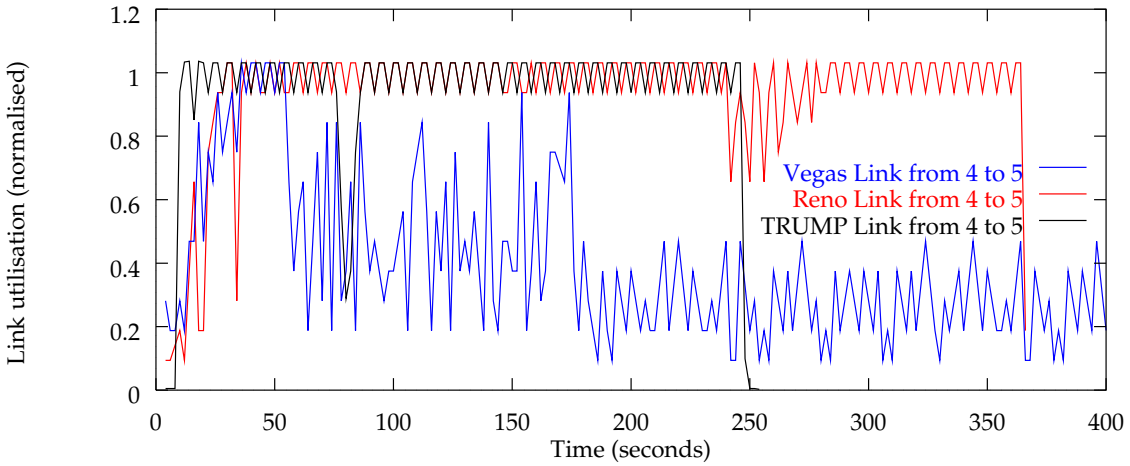


Figure 31: 4 → 5 Link Utilisation in Scenario 3

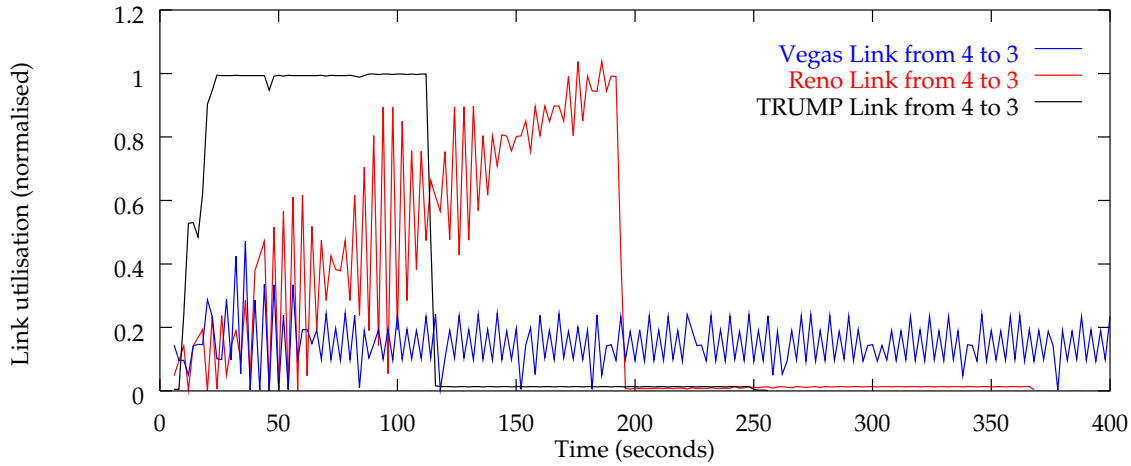


Figure 32: 4 → 3 Link Utilisation in Scenario 3

9.6.4 End-to-End & Round-trip Times

End-to-end times are smooth for TRUMP but not for TCP, as shown for source 6 in Figure 33. Average end-to-end times for TRUMP, Vegas and Reno are 3.41, 3.41, and 4.07 seconds, respectively. Note the large increase in TRUMP's end-to-end delay when router 4 has many packets queued.

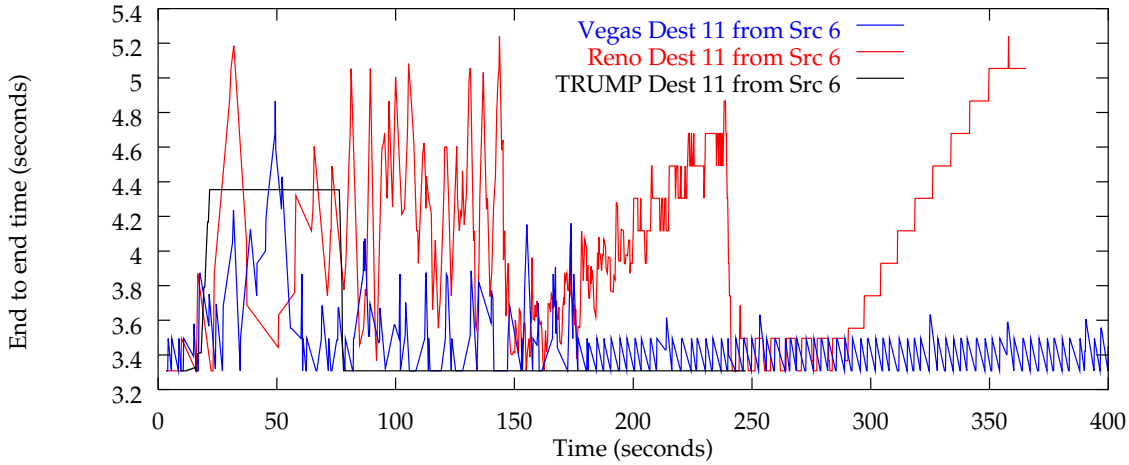


Figure 33: End-to-End Times in Scenario 3

9.6.5 TRUMP Transmission Rates

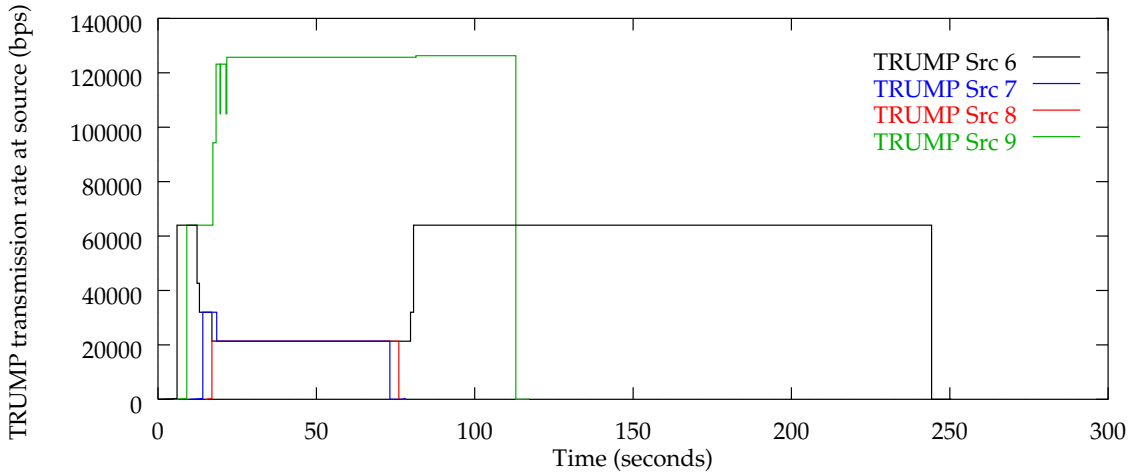


Figure 34: RBCC Rates in Scenario 3

Rates calculated by RBCC are also smooth and have no long-term oscillations. There are small fluctuations in individual rates when traffic flows start and stop, and this is due to the bandwidth-scaling congestion avoidance heuristic in RBCC's implementation.

9.7 Scenario 4

Scenario 4 is shown in Figure 35. This scenario is designed to see how well existing traffic sources react to other sources starting and stopping. Here, the links between routers 17 to 24 take turns being the main bottleneck at different times. The number of sources starting within such a short time should lead to short-term congestion.

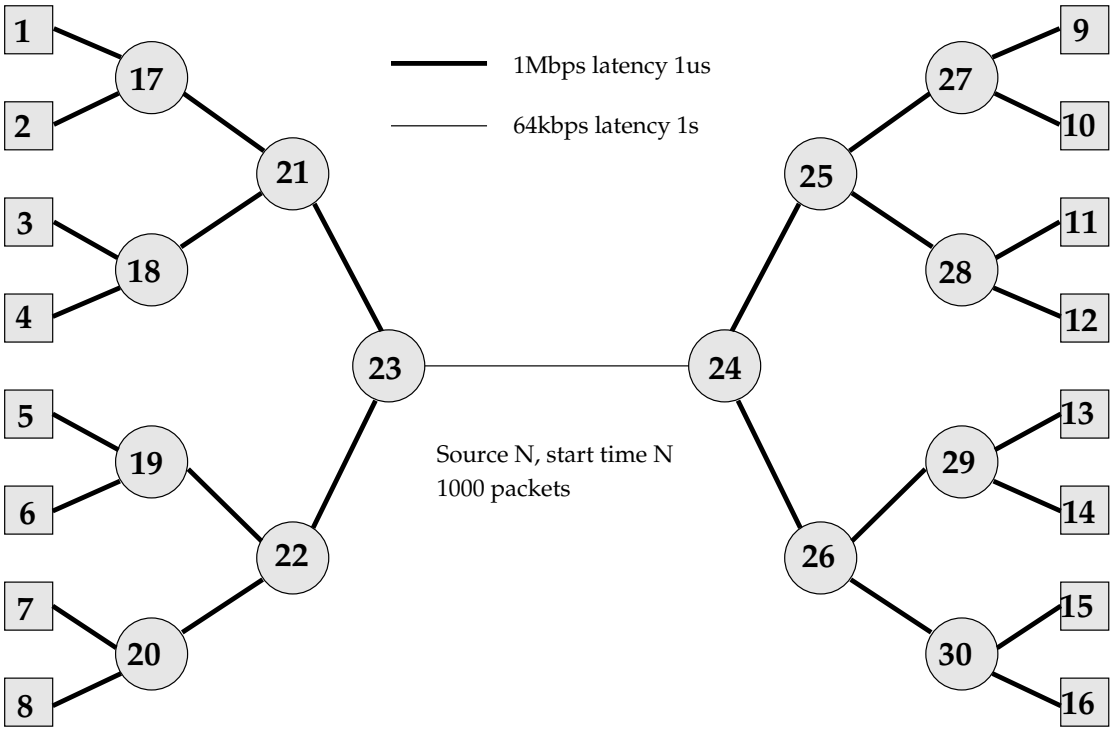


Figure 35: Scenario 4

For TRUMP, the main bottleneck is router 23, although router 21's buffers for the link to 23 peak at 2 packets at times 3 and 4. Router 23's packet buffer to 24 reaches 4 at times 5 and 6, but never exceeds 2 packets queued after time 8; this is shown in Figure 36.

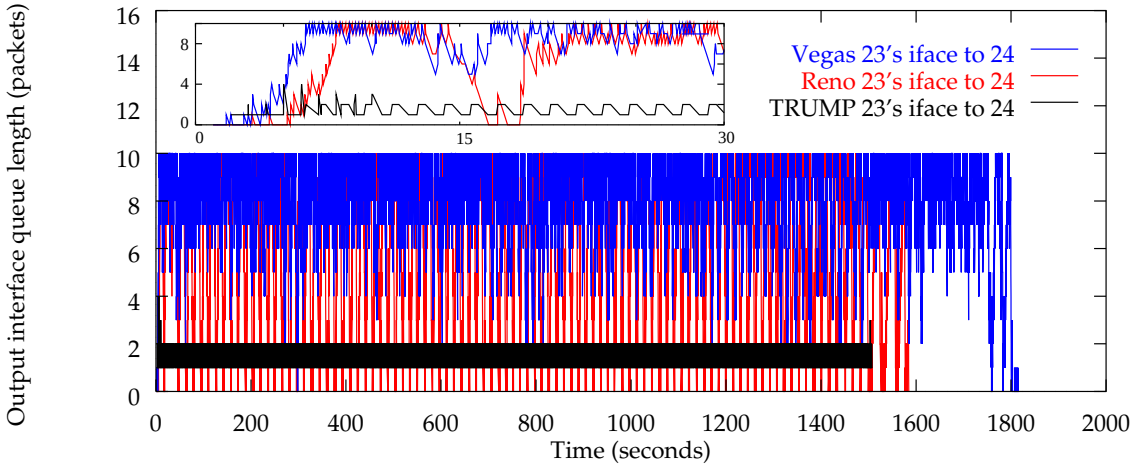


Figure 36: Queue Lengths in Scenario 4

For TCP Reno, the only congested router is 23, and it remains in a severely congested state while any source is transmitting, again shown in Figure 36. The output buffer's queue size fluctuates between 1 and 10 packets, with an average queue length of 6.74 packets. Needless to say, there is a large amount of lost traffic; 782 packets are dropped, spread evenly over all sources. This is approximately 9% packet loss.

TCP Vegas also congests router 23 primarily, as shown in Figure 36, and to the same extent. Compared to Reno, Vegas loses *2,702* packets, spread evenly over all sources. This is approximately 34% packet loss.

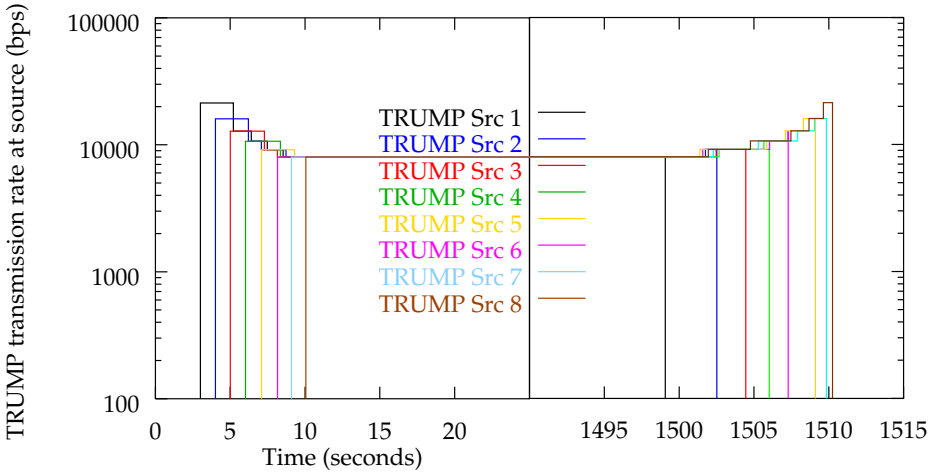


Figure 37: TRUMP Rates in Scenario 4

The transmission rates calculated by RBCC show a lovely smooth reduction in each sources' rate as new sources begin to transmit, as shown in Figure 37. The link 23 $\rightarrow$ 24 is 99% utilised while data transmission is in progress. End-to-end and round-trip times peak to high values initially, and then settle down to near-constant values from time 23 onwards, as is shown for source 6 in Figure 38.

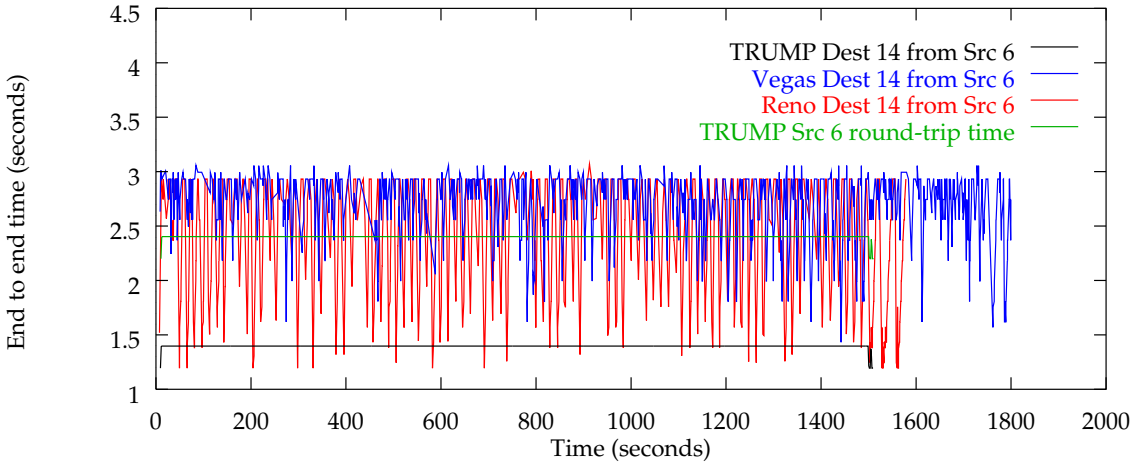


Figure 38: End-to-End and Round-trip Times in Scenario 4

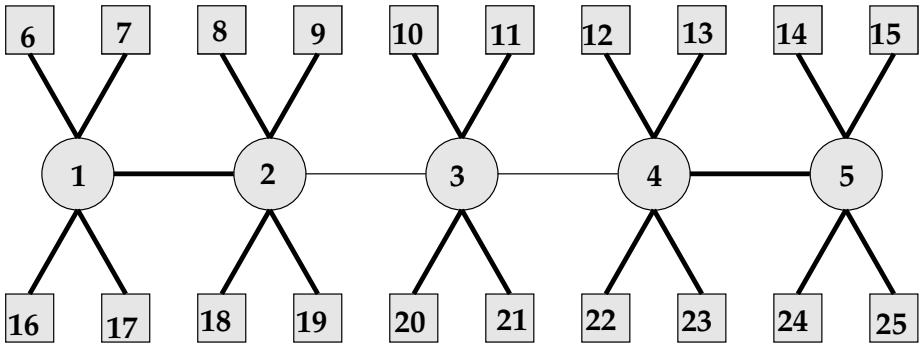
For TCP Reno and Vegas, the link 23 $\rightarrow$ 24 is 93% and 99.7% utilised while data transmission is in progress, respectively. End-to-end and round-trip times are *extremely* peaky for all sources, as is shown for source 6 in Figure 38.

A summary of per-source results for TRUMP, Vegas and Reno is given below. Again note that the averages and deviations for both end-to-end and round-trip times are lower for TRUMP than for the TCP flavours. As noted before, Vegas loses more packets than Reno; it also retransmits many more packets than either Reno or TRUMP.

Source Node	End Time	Avg Rate	End to end Avg	End to end S.Dev	Round-trip Avg	Round-trip S.Dev	Pkts Lost	Pkts Retx'd
TRUMP 1	1497.6	8037	1.30	0.01	2.30	0.01	0	1
Vegas 1	1782.2	10126	2.69	0.33	3.28	0.80	288	496
Reno 1	1498.0	8962	2.43	0.56	3.33	0.26	106	112
TRUMP 2	1501.0	8024	1.42	0.01	2.42	0.01	0	1
Vegas 2	1736.5	10343	2.70	0.30	3.14	0.93	284	491
Reno 2	1529.5	8720	2.44	0.54	3.36	0.24	101	106
TRUMP 3	1503.1	8018	1.36	0.01	2.37	0.01	0	1
Vegas 3	1609.0	11604	2.70	0.31	3.03	0.98	278	550
Reno 3	1500.8	8941	2.42	0.58	3.31	0.23	104	110
TRUMP 4	1504.7	8015	1.29	0.01	2.29	0.01	0	1
Vegas 4	1756.0	9404	2.70	0.31	3.59	0.34	248	370
Reno 4	1458.5	9067	2.46	0.54	3.36	0.21	94	95
TRUMP 5	1506.2	8013	1.53	0.02	2.53	0.02	0	1
Vegas 5	1816.2	9461	2.62	0.44	3.40	0.62	269	419
Reno 5	1433.8	9213	2.44	0.57	3.32	0.20	94	95
TRUMP 6	1508.2	8016	1.40	0.02	2.40	0.02	0	2
Vegas 6	1800.7	9408	2.69	0.32	3.45	0.57	265	403
Reno 6	1575.6	8456	2.43	0.55	3.34	0.30	96	96
TRUMP 7	1508.9	8017	1.22	0.00	2.22	0.00	0	2
Vegas 7	1757.8	9904	2.71	0.29	3.51	0.48	278	439
Reno 7	1401.5	9397	2.44	0.56	3.34	0.17	90	90
TRUMP 8	1509.5	8019	1.56	0.03	2.56	0.03	0	2
Vegas 8	1796.0	11154	2.69	0.31	2.67	1.02	314	656
Reno 8	1584.2	8458	2.37	0.59	3.33	0.36	97	97

9.8 Scenario 5

Scenario 5 is shown in Figure 39. This simulation models a number of traffic flows, some of which flow across different sections of the 1M, 100k, 64k, 1M low-speed link. A congestion control scheme should not penalise one traffic flow for passing through more intermediary routers than another traffic flow.



6 -> 10 start 0s
7 -> 24 start 20s
8 -> 21 start 50s
9 -> 25 start 60s
10 -> 23 start 61s
11 -> 16 start 1s
12 -> 19 start 21s
13 -> 20 start 40s
14 -> 22 start 41s
15 -> 17 start 49s

1-> 2 1Mbps latency 5ms
2 -> 3 100kbps latency 8ms
3 -> 4 64kbps 800ms
4 -> 5 1Mbps 10ms

Figure 39: Scenario 5

With TRUMP, several routers are congested at different times. Router 3 on the link to 2 has queue sizes above 2 from time 24 to 164, with three peaks to 5 packets queued around time 58. Router 2 on the link to 3 has short peaks to 3 packets queued from time 60 to 83. Router 4 on the link to 3 has two peaks to 3 packets queued at times 57 and 61. Router 3 on the link to 4 has peaks above 2 packets queued from time 61 to time 447. Some of these queue sizes are shown in Figure 40. No packets are lost by any RBCC routers.

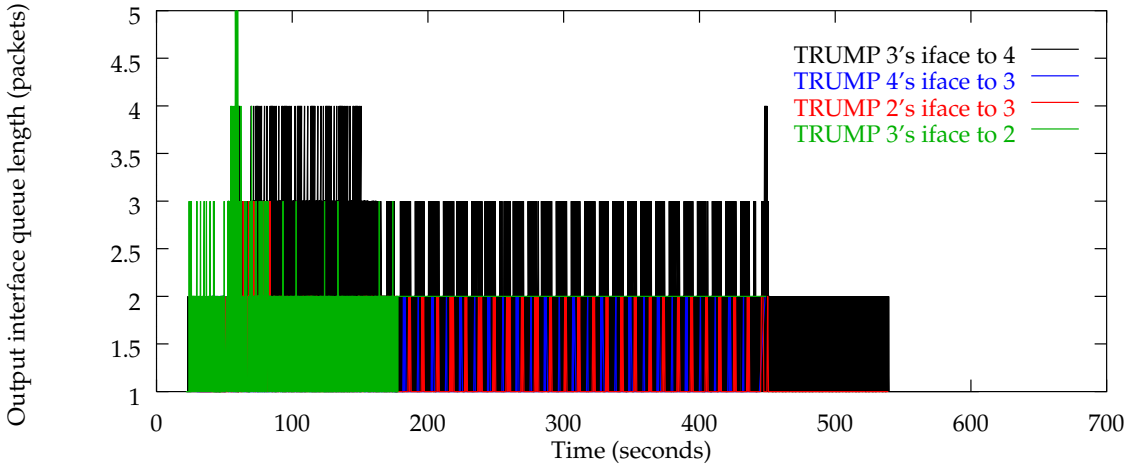


Figure 40: Queue Sizes in Scenario 5 by TRUMP

The rates calculated by RBCC are mostly smooth, with some small fluctuations for most sources during the times 20 to 80; these are due to the packet queue sizes crossing the RBCC threshold in some routers. The rates used by each TRUMP source are shown in Figure 41.

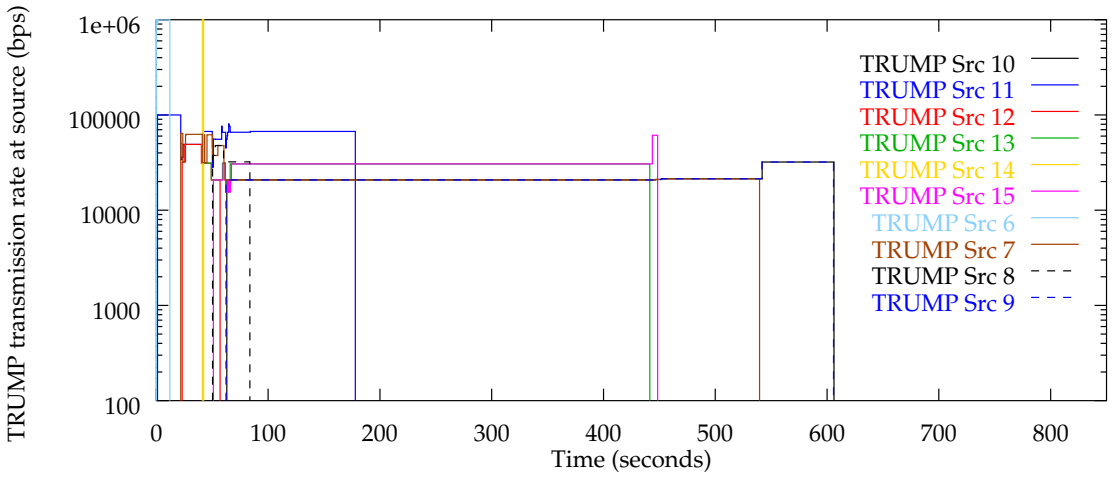


Figure 41: TRUMP Rates in Scenario 5

Average link utilisations for the low-speed links are given in the following table. The end-to-end times for TRUMP fluctuate a lot until time 80 and then show little deviation, except for sources 16 and 17. Round-trip times show some deviation for all sources at all times.

Link	TRUMP	TCP Vegas	TCP Reno
$2 \rightarrow 3$	0.43	0.37	0.41
$3 \rightarrow 2$	0.43	0.37	0.39
$4 \rightarrow 3$	0.70	0.79	0.60
$3 \rightarrow 4$	0.98	0.93	0.85

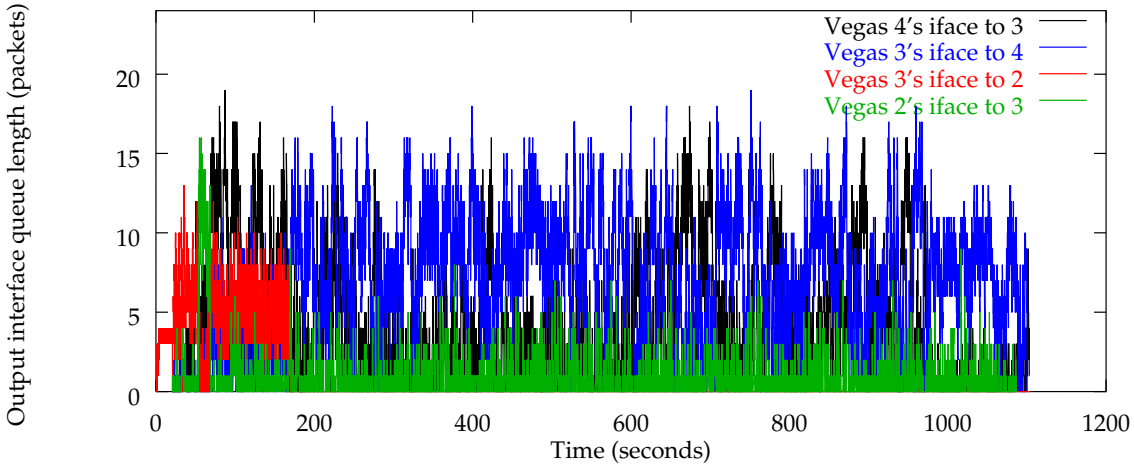


Figure 42: Queue Sizes in Scenario 5 by Vegas

In Scenario 5, TCP Vegas exhibits severe congestion and packet loss in routers 1, 2, 3, 4 and 5, with long-term queue lengths above 10 packets, as shown in Figure 42. 140 packets were lost, mainly by routers 3 and 4, with sources 7, 9, 10 and 15 losing the majority.

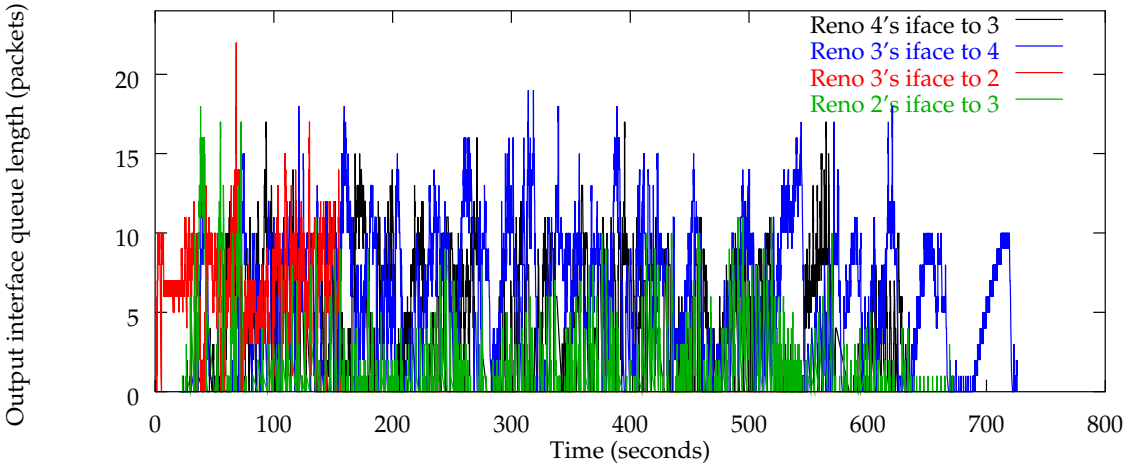


Figure 43: Queue Sizes in Scenario 5 by Reno

TCP Reno exhibits even worse congestion and packet loss in routers 1, 2, 3, 4 and 5 than Vegas, with long-term queue lengths also above 10 packets, as shown in Figure 43. 405 packets were lost, mainly by routers 3 and 4, with all sources except 16, 19, 20, 24 and 25 losing more than 10 packets.

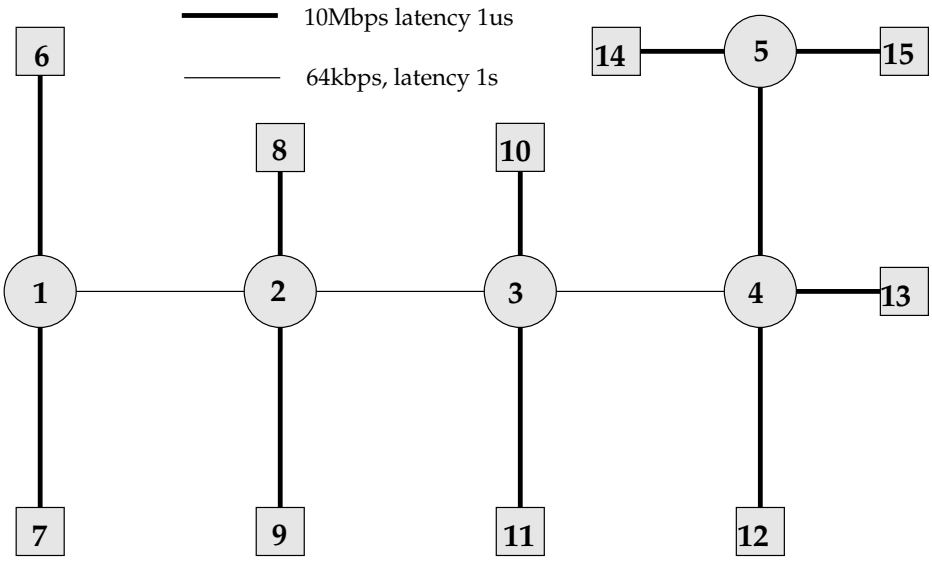
The end times for all sources are given in the following table. Although some Vegas sources finish earlier than TRUMP (sources 8 and 11), many finish significantly later than TRUMP (sources 7, 9, 10, 12, 13 and 15). This demonstrates that TCP Vegas treats long-haul connections unfairly, giving more bandwidth to short round-trip time connections.

Source Node	End Time	Avg Rate	End to end Avg	End to end S.Dev	Round-trip Avg	Round-trip S.Dev	Pkts Lost	Pkts Retx'd
TRUMP 6	12.1	997051	0.02	0.00	0.03	0.00	0	2
Vegas 6	12.1	1000324	0.05	0.00	0.06	0.00	0	0
Reno 6	16.0	780865	0.05	0.04	0.09	0.02	0	31
TRUMP 7	539.1	23263	1.17	0.03	2.17	0.12	0	3
Vegas 7	1086.5	13097	2.13	0.52	3.33	0.55	33	155
Reno 7	671.4	20576	2.29	0.66	3.39	0.46	132	108
TRUMP 8	83.3	36418	0.16	0.05	0.27	0.08	0	0
Vegas 8	67.6	72338	0.80	0.28	0.95	0.11	0	0
Reno 8	74.7	73494	0.61	0.34	1.09	0.15	30	39
TRUMP 9	605.3	22205	1.43	0.13	2.38	0.21	0	5
Vegas 9	1066.1	24940	2.08	0.48	1.33	0.77	37	1087
Reno 9	604.8	23258	2.16	0.55	3.47	0.38	51	49
TRUMP 10	605.8	22213	1.15	0.02	2.01	0.06	0	5
Vegas 10	1102.6	24713	1.78	0.45	0.97	0.25	59	1135
Reno 10	725.6	20312	1.81	0.57	2.86	0.46	65	114

Source Node	End Time	Avg Rate	End to end Avg	End to end S.Dev	Round-trip Avg	Round-trip S.Dev	Pkts Lost	Pkts Retx'd
TRUMP 11	177.8	67973	0.19	0.04	0.25	0.08	0	1
Vegas 11	170.5	71433	0.60	0.18	0.69	0.14	0	5
Reno 11	156.3	81758	0.81	0.27	0.87	0.18	25	50
TRUMP 12	56.4	37448	1.15	0.08	2.17	0.11	0	4
Vegas 12	127.4	21318	2.25	0.50	1.98	1.34	5	85
Reno 12	147.3	11398	2.31	0.54	3.53	0.46	21	17
TRUMP 13	441.0	30220	0.99	0.01	2.02	0.12	0	5
Vegas 13	976.0	27527	1.55	0.46	1.02	0.45	27	1141
Reno 13	465.9	30150	1.66	0.51	3.10	0.37	31	60
TRUMP 14	42.2	1008221	0.02	0.00	0.04	0.00	0	2
Vegas 14	42.3	1003458	0.06	0.01	0.07	0.01	0	0
Reno 14	49.6	934329	0.10	0.01	0.05	0.02	0	24
TRUMP 15	448.2	30539	1.18	0.04	2.39	0.15	0	11
Vegas 15	1105.2	24529	1.75	0.50	1.20	0.33	32	1152
Reno 15	633.9	22035	1.87	0.58	3.45	0.39	73	67

9.9 Scenario 6

Scenario 6 is shown in Figure 44:



6 -> 15, start 0s, 1000 pkts

7 -> 8, start 1s, 1000 pkts

9->10, start 75s, 1000 pkts

11 -> 12, start 110s, 1000 pkts

13 -> 14, start 145s, 1000 pkts

Figure 44: Scenario 6

This scenario models a long-haul traffic flow, $6 \rightarrow 15$, which must compete against a number of short-haul cross-traffic flows. The end times for all sources are given below.

Source Node	End Time	Avg Rate	End to end Avg	End to end S.Dev	Round-trip Avg	Round-trip S.Dev	Pkts Lost	Pkts Retx'd
TRUMP 6	315.0	40894	4.84	0.12	8.86	0.12	0	46
Vegas 6	486.0	26246	4.93	0.39	8.88	0.75	1	20
Reno 6	503.5	25191	5.18	0.44	9.17	0.35	11	14
TRUMP 7	79.5	32157	1.19	0.00	2.20	0.00	0	5
Vegas 7	100.4	41549	1.56	0.33	1.54	0.76	0	138
Reno 7	1308.1	6610	1.33	0.39	2.62	0.47	0	63
TRUMP 9	153.6	32157	1.19	0.00	2.19	0.00	0	5
Vegas 9	168.5	44023	1.68	0.44	1.53	0.83	0	138
Reno 9	151.2	36528	1.89	0.54	2.75	0.33	11	20
TRUMP 11	189.0	32156	1.27	0.09	2.28	0.09	0	6
Vegas 11	204.3	46296	1.54	0.29	1.34	0.72	0	155
Reno 11	303.8	41912	1.53	0.38	2.42	0.39	19	31
TRUMP 13	223.9	32156	1.37	0.09	2.37	0.09	0	6
Vegas 13	201.9	45570	1.75	0.36	2.70	0.42	0	3
Reno 13	632.5	17057	1.39	0.37	2.63	0.49	26	45

As can be seen, source 6 is penalised by both TCP versions because it is a long-haul traffic flow. The short-haul traffic flows end much faster in TCP Vegas than in TRUMP, indicating that they are favoured.

With TRUMP, the only congested link is $4 \rightarrow 5$, with router 4's queue oscillating between 1 and 2 packets queued from time 156 to time 224. The transmission rates calculated by RBCC are smooth. End-to-end and round-trip times for TRUMP/RBCC are surprisingly flat, despite the congestion. In comparison, TCP Reno and Vegas exhibit network congestion in several routers, and some packets are lost; this is shown in Figure 45.

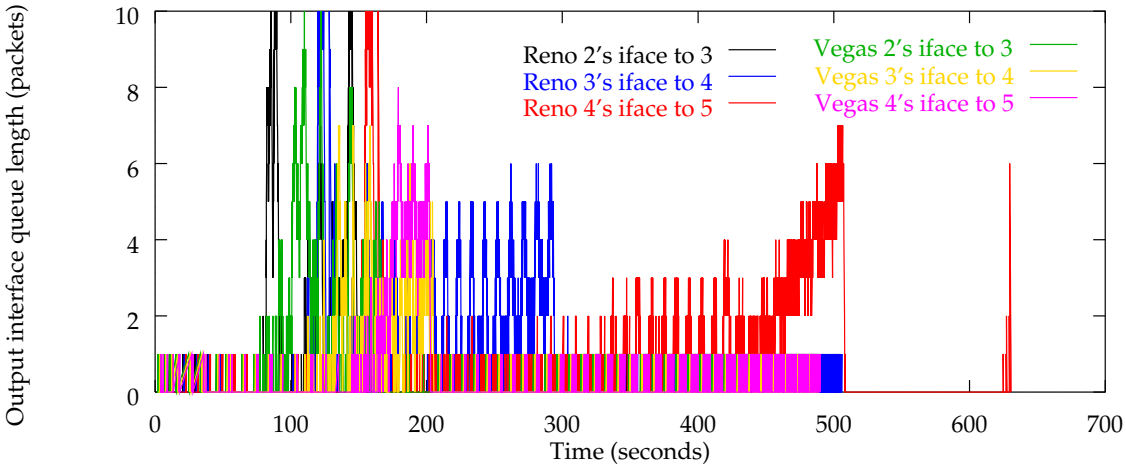


Figure 45: TCP Queue Lengths in Scenario 6

9.10 Scenario 7

Scenario 7 is the same as Scenario 2. However, in this scenario, the traffic flows vary their packet sizes evenly across a range of sizes:

Flow	Range
$5 \rightarrow 10$	[70:1500]
$6 \rightarrow 11$	[400:900]
$7 \rightarrow 12$	[700:1500]
$8 \rightarrow 9$	[900:1500]

Unfortunately, the version of TCP Reno and TCP Vegas that are available in REAL 4.0 do not have the ability to alter their packet sizes on the fly, so no comparisons can be drawn with TRUMP/RBCC and TCP in this scenario.

The usual per-source results for Scenario 7 are given in the following table.

Source Node	End Time	Avg Rate	End to end Avg	End to end S.Dev	Round-trip Avg	Round-trip S.Dev	Pkts Lost	Pkts Retx'd
TRUMP 5	207.2	58939	1.16	0.07	2.20	0.09	0	8
TRUMP 6	168.2	49679	1.19	0.07	2.22	0.08	0	8
TRUMP 7	260.1	49606	1.20	0.06	2.26	0.08	0	17
TRUMP 8	274.4	47842	1.15	0.02	2.28	0.08	0	13

Packet loss is nonexistent, and end-to-end and round-trip deviation is low. Packet queue lengths in the bottleneck router is higher than for Scenario 2, and fluctuates more, as shown in Figure 46. This is a consequence of the sizes and earlier arrival times for the smaller packets. This scenario shows that a congestion avoidance mechanism which relies simply on the number of packets queued will sometimes incorrectly perceive congestion.

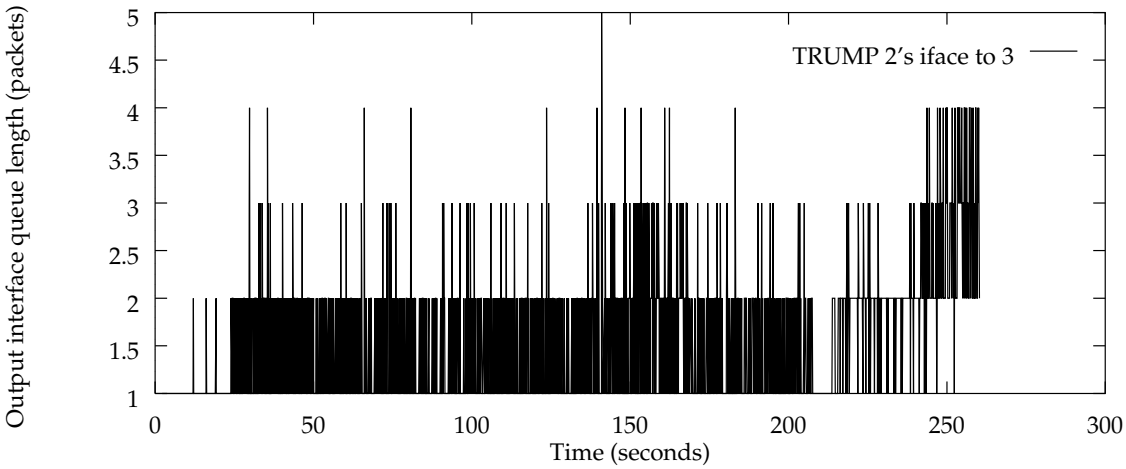


Figure 46: TRUMP Queue Lengths in Scenario 7

The effect of the small packets and the larger router queue sizes has little effect on the transmission rates passed back by RBCC to the traffic sources, as shown in Figure 47. The rate are, as usual, stable, but with a number of dips where the RBCC congestion avoidance mechanism has fired. Link utilisation is essentially unchanged from Scenario 2.

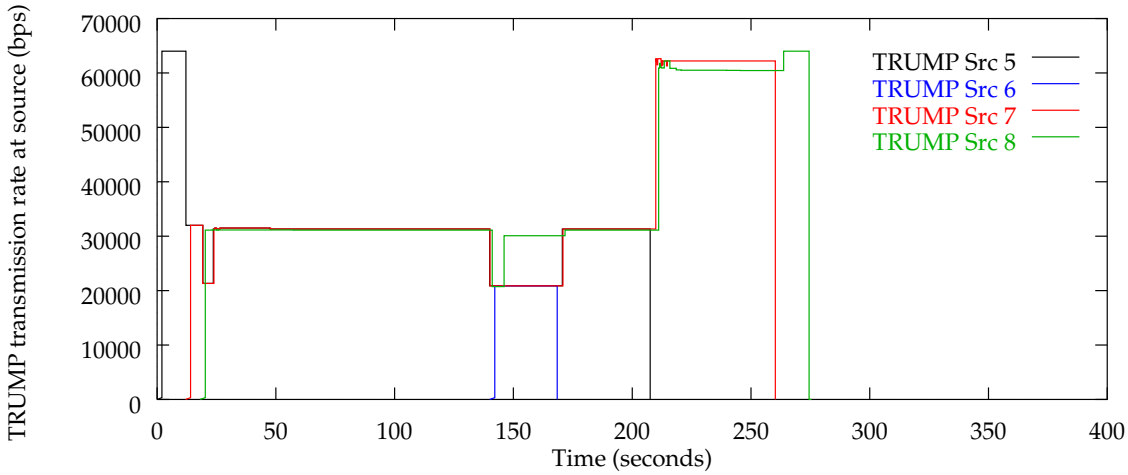


Figure 47: TRUMP Source Rates in Scenario 7

9.11 Scenario 8

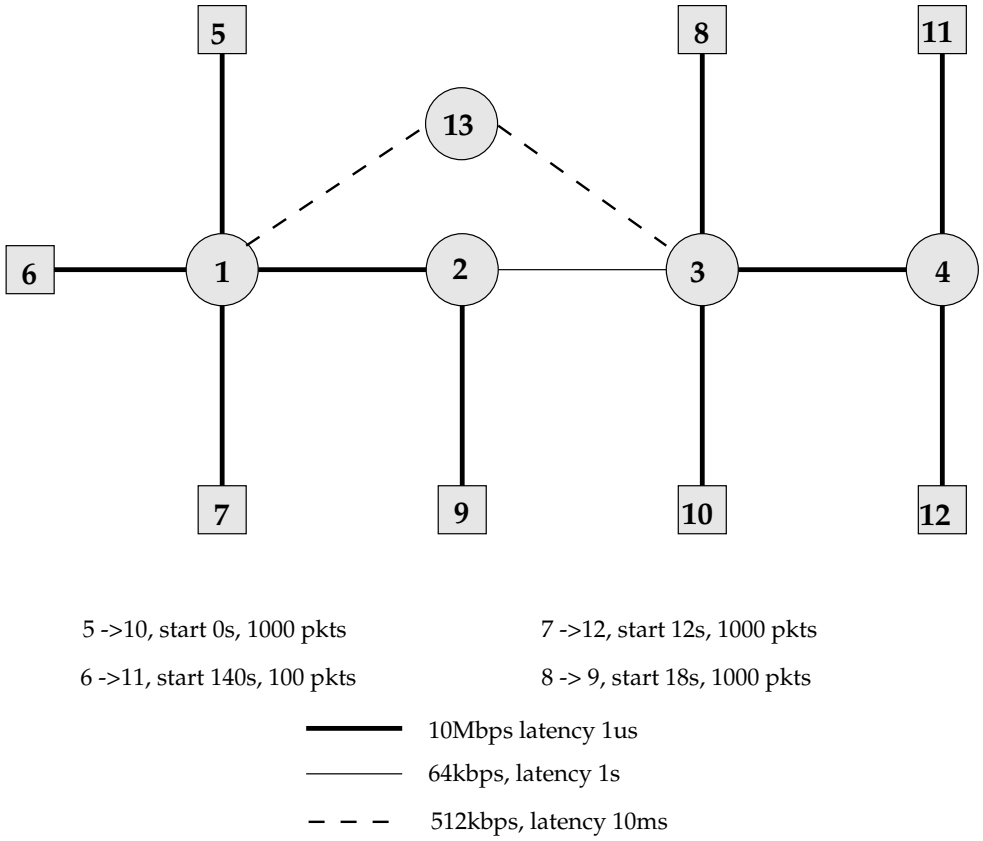


Figure 48: Scenario 8

Scenario 8 is shown in Figure 48. This is similar to Scenario 2, but there is an extra router, 13, and two extra links which connect router 13 to routers 1 and 3. Initially, the new links and router are not used. At time 100, router 1 changes its routing table to forward packets for destination 10 via router 13. This affects the data flow of source 5. The new route via router 13 offers more bandwidth than the original route ($1 \rightarrow 2 \rightarrow 3$).

As flow 6 → 11 doesn't come online until time 140, only the flows from sources 5, 7 and 8 are affected at the route change time of 100 seconds. The route change gives the flow 5 → 10 much more bandwidth; it should be able to use the full 512kbps across links
(1 → 13 → 3). Flow 7 → 12 should then be able to use the bandwidth previously used by flow 5 → 10. Let us examine the route change on TRUMP, Vegas and Reno in turn.

The first TRUMP data packet from source 5 that crosses the new route has its **Rate** field updated with the new bandwidth, and this value reaches source 5 at time 100.24. The new rate calculated by RBCC is exactly 512kbps. Router 2 detects the lost of flow 5 → 10 at time 105.0, as described in Section 8.4. It calculates a new rate for source 7, which it receives at time 107.6. The new rate is 62,361bps, which provides for the acknowledgments of flow 8 → 9. The rates calculated by RBCC are shown in Figure 49.

TCP Vegas begins to open its window, prompted by round-trip time reductions, at time 100.4, and Reno begins to open its window at time 101.1, as shown in Figure 50.

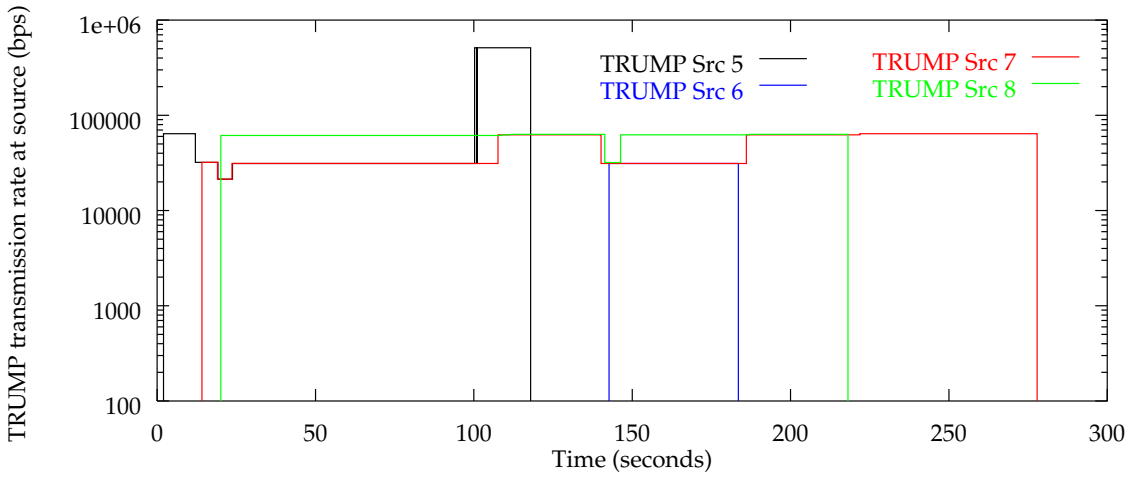


Figure 49: RBCC Rates in Scenario 8

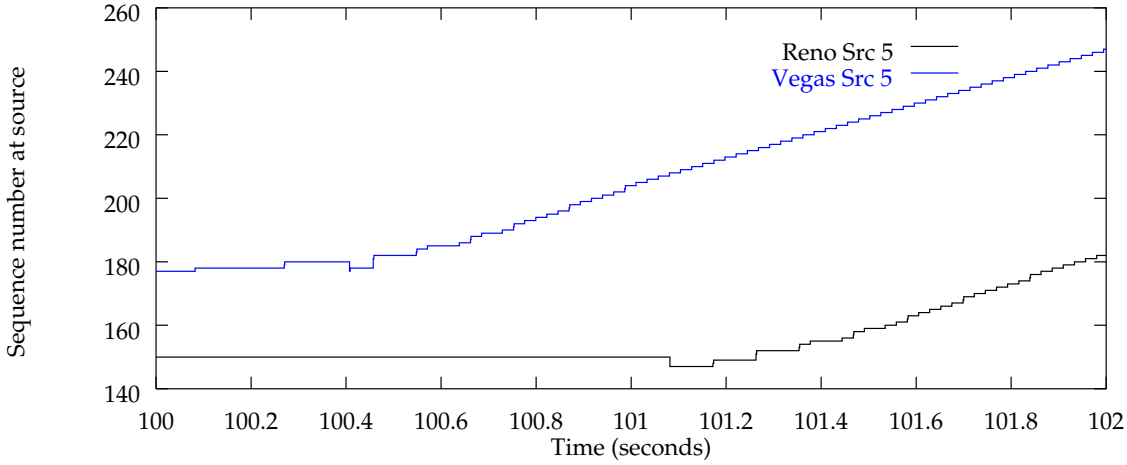


Figure 50: TCP Source 5 Sequence Numbers in Scenario 8

What is difficult to explain is the apparent transmission rate of source 7 in both TCP Vegas and Reno. Optimally, source 7's rate should double at the route change time of 100 seconds, halve when source 6 comes online at time 140 seconds, and redouble when source 6 finishes

transmission. Reno appears to show this behaviour, as seen in the sequence number diagram in Figure 51; however, the initial rate is kept low by the Slow Start mechanism. Vegas seems to reach a stable rate just before time 50 seconds, increases its rate after the route change, slows down in response to source 6, but then never regains its original rate once source 6 finishes transmitting. This behaviour was also seen in Scenario 2, and appears to be a problem in TCP.

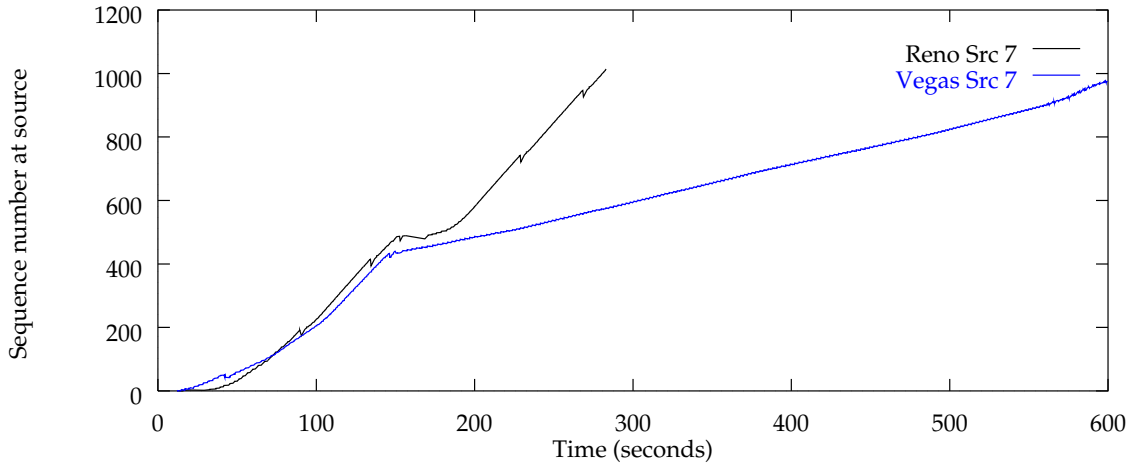


Figure 51: TCP Source 7 Sequence Numbers in Scenario 8

The following table gives the per-flow results for Scenario 8. Overall, source 5 goes faster for Vegas than TRUMP/RBCC, but Vegas loses packets. Despite a 5-second delay to detect the route change by router 2, TRUMP's source 7 still achieves a higher rate than either TCP, with no packet loss. And as with Scenario 2, source 8 achieves a much higher rate with TRUMP than either version of TCP.

Source Node	End Time	Avg Rate	End to end Avg	End to end S.Dev	Round-trip Avg	Round-trip S.Dev	Pkts Lost	Pkts Retx'd
TRUMP 5	118.2	103793	0.42	0.56	0.74	1.03	0	5
Vegas 5	119.8	110008	0.56	0.79	0.50	0.80	5	91
Reno 5	121.3	107848	0.53	0.74	0.56	0.86	88	77
TRUMP 6	182.7	31465	1.49	0.02	2.58	0.06	0	6
Vegas 6	224.3	33892	2.07	0.45	1.02	0.46	4	134
Reno 6	179.9	37852	1.80	0.54	2.64	0.34	7	12
TRUMP 7	275.9	46344	1.22	0.06	2.30	0.09	0	11
Vegas 7	599.7	33201	1.57	0.40	1.68	1.12	7	647
Reno 7	283.1	45282	2.03	0.59	2.98	0.49	7	8
TRUMP 8	218.1	61402	1.19	0.00	2.38	0.10	0	12
Vegas 8	484.6	38266	1.51	0.23	1.95	0.94	0	474
Reno 8	599.1	11688	1.35	0.39	2.63	0.43	30	33

9.12 Scenario 9

Scenario 9 is a modification to Scenario 6 (Figure 44), but with the 7 $\rightarrow$ 8 cross-flow starting at time 10, after source 6 learns of its initial rate. The links 1 to 4 impose a 3-second delay in the feedback of rate information to source 6 from its destination.

Section 7.1 suggests that, although selective acknowledgments lower the network load due

to acknowledgment traffic, the effect of acknowledgment delays may induce higher network congestion. This scenario examines the effect of large selective acknowledgment sizes, combined with Rate Quenches, on the framework.

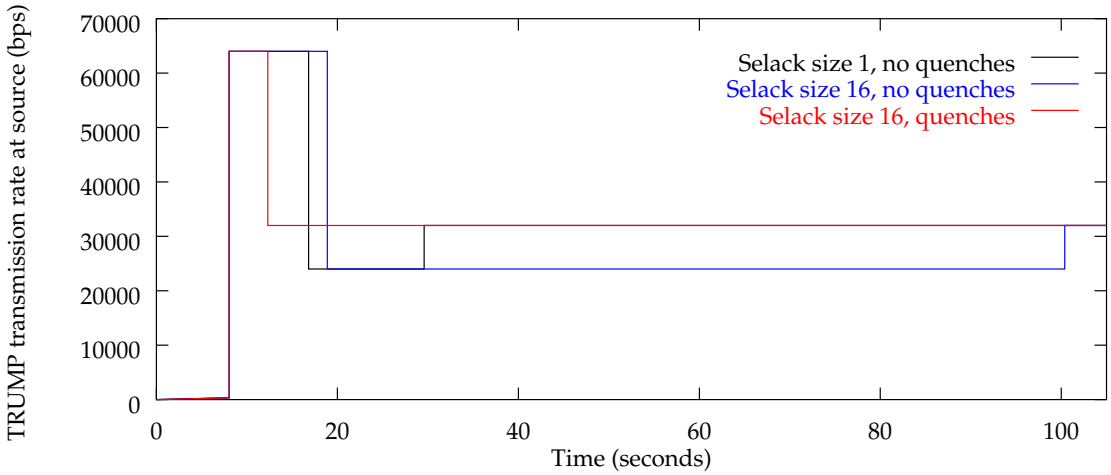


Figure 52: Source 6's Initial Rate Changes in Scenario 9

Figure 52 shows the rates used by TRUMP for source 6, for several selective acknowledgment sizes and Rate Quench combinations. For all three plots, source 6 obtains an initial rate at time 8 seconds.

The black plot shows source 6 using a selective acknowledgment size of 1, with no Rate Quenches being generated by the network. Source 7 starts at time 10, and requires some of the bandwidth in use by source 6. RBCC recalculates a new rate for source 6, and this arrives at the source at time 16.8 seconds. Because of the large round-trip time, the packet queue in router 1 exceeds the congestion avoidance threshold at time 13.6 seconds, and two packets for source 6 are lost around time 16 seconds. The link-delayed new rate for source 6 arrives at the source at time 17 seconds, and has been scaled by 0.75 by the network's congestion avoidance scheme. It is only after router 1's queues have fallen, at time 29, that source 6 receives its RBCC rate allocation unscaled.

The blue plot shows source 6's operation with a selective acknowledgment of 16, with no Rate Quenches being generated by the network. The effect of the larger acknowledgment size is to delay the reception of a new rate for source 6 until time 19 seconds. Again, because of router queue sizes, this has been scaled by the network's congestion avoidance scheme. The time until the avoidance scheme terminates scaling is also delayed, both by the selective acknowledgment delays and by the time to drain router 1's buffers.

The red plot shows source 6's operation with a selective acknowledgment of 16, and with Rate Quenches being generated by the network. The effect of Rate Quenches is dramatic. A new rate for source 6 is returned at time 12 seconds, and is not scaled as the buffer occupancy in router 1 has not exceed the congestion avoidance threshold.

Scenario 9 shows that high round-trip times do slow the congestion framework's reaction time, and delays due to large selective acknowledgment sizes increases this reaction time further. Rate Quenches appear to be a useful mechanism to circumvent these delays.

9.13 Summary

The simulation of nine hand-crafted network scenarios in this chapter show that the proposed rate-based congestion framework, with TRUMP and RBCC as component functions, appears to work very well at congestion control. Let us review the congestion characteristics for both TRUMP/RBCC and for the two flavours of TCP.

Packet Losses and Retransmissions Overall, TRUMP loses *no* packets in any of the scenarios. Vegas loses many fewer packets than Reno, except for Scenario 4 where Vegas loses over 3 times as many packets as Reno. However, Vegas and Reno loses 2,956 and 1,563 packets, respectively. On this basis alone, TRUMP/RBCC provides better congestion control than TCP.

The amount of packet retransmission varies greatly between scenarios, but in general TRUMP's level of packet retransmission is roughly of the same magnitude as TCP Reno's. Vegas, on the other hand, appears to aggressively retransmit packets, which indicates that its timeout strategy is still incorrect. Note that TRUMP does not retransmit packets because of packet losses: instead, it lazily retransmits packets until their acknowledgments are received. This is done to avoid setting a round-trip timer when there is no new data to transmit.

Router Queue Lengths TRUMP/RBCC generally keeps router queue lengths down around 1 or 2 packets, except where new traffic flows came on line: the delay to return new sustainable rates to all traffic flows allows router queue lengths to grow. The primitive 'threshold' congestion avoidance scheme in RBCC is adequate to bring the queue levels back down to 1 or 2 packets. In contrast, both TCP versions keep queue lengths in bottleneck routers above 2 packets most of the time. This adds to the delay of packet transmission, and makes each router more susceptible to short-term congestion which fills the remaining buffer space and causes lost packets. On the basis of router queue lengths, TRUMP/RBCC appears to give good congestion control.

End-to-end & Round-trip Times The higher router queue lengths in TCP increases the average end-to-end and round-trip times for traffic flows. Due to the lower router queue lengths in TRUMP/RBCC, average end-to-end and round-trip times are also lower. The 'bursty' window size increases made by both TCP flavours increases router queue lengths in the short-term, which adds variance to end-to-end and round-trip times. Ironically, this in turn makes TCP's round-trip time estimation more difficult. Not only does TRUMP/RBCC not require a round-trip timer, but its Leaky Bucket packet admission works to reduce variance in end-to-end and round-trip times.

Although this feature is not directly related to congestion control, it is a boon to applications such as voice and video transmission where low end-to-end time variance is desired.

Link Utilisation and RBCC Rate Allocations RBCC appears to derive near-optimal transmission rates for all traffic sources. This keeps 'bottleneck' links operating at near 100% utilisation. TCP's Slow Start and bandwidth-probing strategies either keeps bottleneck links operating either below their capacity, or offers too much load for links to transmit. The result is either poor utilisation or packet loss. As well, for some scenarios, TCP did not probe and find extra bandwidth when it became available. The reason for this behaviour is unknown.

Scenario Variations Finally, the proposed congestion control framework, with TRUMP and RBCC as component functions, was able to cope with finite desired rates from traffic sources, ranges of packet sizes from sources, and low-frequency route changes.

Chapter 10

Simulating Random Scenarios

10.1 Introduction

The performance of the rate-based congestion framework using TRUMP/RBCC has been compared against TCP Reno and TCP Vegas in a set of hand-designed network scenarios with specific congestion problems. It may be argued that this small set of scenarios does not fully test the framework, and that a wider range of scenarios should be tried.

In this chapter I discuss the testing of the framework on a set of 500 randomly generated network scenarios. Both TCP Reno and TCP Vegas are also simulated on these scenarios to allow for comparisons of simulation results.

10.2 Scenario Generation

The 500 network scenarios were pseudo-randomly generated[1] by a small Perl script. Each scenario contains 9 routers, 15 sources and 15 destinations. The 9 routers are randomly interconnected to form a single network, and the 15 sources and destinations are randomly connected to the routers.

Each source transmits 1,500 packets of 1,500 octets to its destination, and the start time for each source is distributed evenly over the interval 0:100 seconds.

In order to increase the probability of network congestion, the overall bit rates and latencies for the links in each scenario were chosen as follows. There is a probability of 0.6 that a link is *fast*: fast links have capacities spread evenly over 1,000,000:10,000,000 bits per second and latencies spread evenly over 100:10,000 microseconds. If a link is not fast, it is *slow*: slow links have capacities spread evenly over 32,000:250,000 bits per second and latencies spread evenly over 10:5,000 milliseconds.

All other network scenario parameters are the same as the values given in Section 9.1. Details on how to obtain the 500 generated network scenarios are given in Appendix G.

[1] The 4.4BSD `rand()` function was used, which conforms to the ANSI C standard. For each scenario generated, it was seeded with the current time in seconds since 1970. There was a delay of at least 100 seconds between the generation of each scenario.

10.3 Abstracted Measurements

As the amount of data obtained from the simulation of 500 network scenarios is overwhelming, a small set of measurements were abstracted from each simulation which describe the level of network congestion and performance in the simulated scenario.

The set of abstracted measurements are:

- The average buffer queue length (in packets) for the router with the highest average length.
- The average buffer queue length (in packets) for all the routers in the simulated scenario.
- The total number of packets lost in the simulated scenario.
- The average utilisation (normalised) for the link with the highest utilisation.
- The effective bit rate of all sources.
- The end-to-end standard deviation for the source with the highest end-to-end standard deviation.
- The inter-packet spacing (in microseconds) as seen by the destinations of traffic flows.
- The improvement to RBCC when packet congestion fields are cached.

10.4 Highest Average Buffer Queue Length

In each simulation there is a router which has the highest average buffer queue length. This router is the most congested node in the network. A perfect congestion control scheme will keep the average queue length for even the most congested node at 1 packet. Figure 53 shows the distribution of the highest average queue lengths.

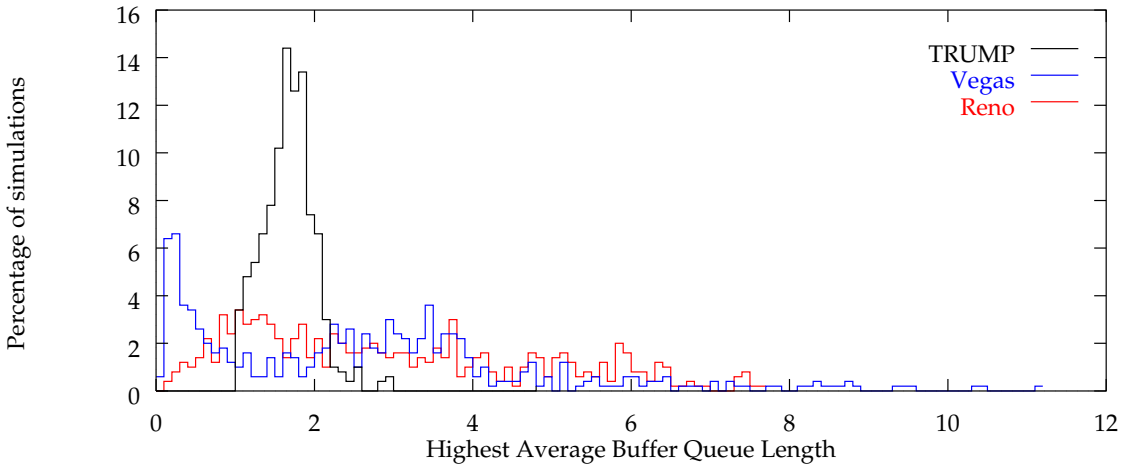


Figure 53: Highest Average Buffer Queue Length

In the 500 scenarios simulated, TRUMP/RBCC keeps the average queue length close to 1 packet. The median of TRUMP/RBCC's distribution is 1.69, and 90% of the distribution falls in the range 1.12:2.13 queued packets. The distribution of highest queue lengths for Reno and Vegas is more widely spread. The median of Reno's distribution is 2.62, and 90% of the distribution falls in the range 0.45:5.99 queued packets. The median of Vegas' distribution is 2.46, and 90% of the distribution falls in the range 0.05:5.43 queued packets.

Clearly, TRUMP/RBCC is attempting to keep bottleneck routers near an optimum queue length of 1, thus ensuring that bottleneck links are utilised but not congested. TCP, on the other hand, either does not attempt to utilise bottleneck links, or overloads them, resulting in congestion. The results for packet loss and link utilisation, given below, reinforce this result.

10.5 Average Buffer Queue Length

The distribution of average queue lengths per simulation is shown in Figure 54. Even for non-bottleneck routers, TRUMP/RBCC attempts to keep queue lengths close to 1 packet, with a distribution median of 0.79 and with 90% of the distribution in the range 0.43:1.12 queued packets.

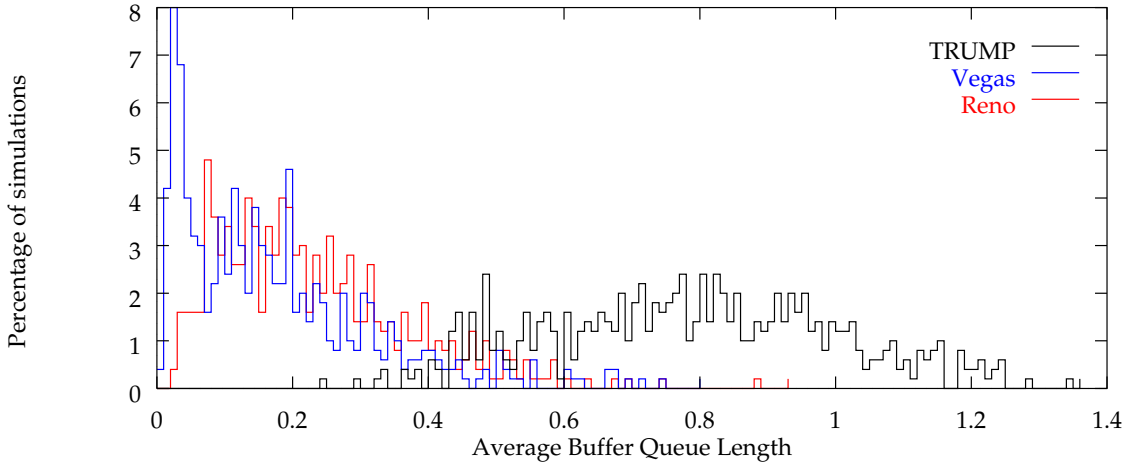


Figure 54: Average Buffer Queue Length

Surprisingly, the two TCP distributions fall below that of TRUMP/RBCC. Reno's distribution median is 0.20 with 90% of the distribution in the range 0.03:0.43 queued packets, and Vegas' distribution median is 0.14 with 90% of the distribution in the range 0.01:0.38 queued packets. It shows that TCP does not make full utilisation of the network's available capacity, and so routers are often under-utilised. Again, this assertion is reinforced by the link utilisation results given below.

10.6 Total Packets Lost

The total packets lost in each simulation demonstrates quite clearly if the simulated network is suffering congestion, and to what extent. A perfect congestion control scheme will prevent any packets from being lost.

Each of the 500 randomly-generated scenarios has 9 routers and 15 traffic flows generating 1,500 packets. Therefore, there must be at least $500 * 15 * 1500 = 11,250,000$ data packets to be sent in the set of 500 scenarios. For TCP and TRUMP, each data packet is acknowledged. Therefore, the 500 scenarios must deliver at least 22,500,000 packets, not including handshaking (connection setup) packets, and retransmissions.

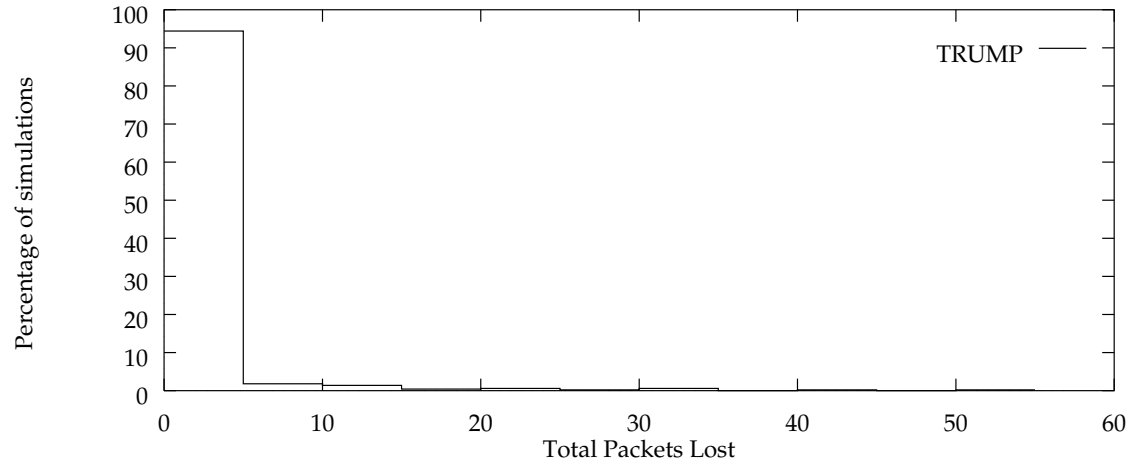


Figure 55: Total Packets Lost by TRUMP/RBCC

The distribution of total packets lost per simulation by TRUMP/RBCC is shown in Figure 55. This is an excellent distribution for packet loss, skewed towards 0 packets lost per scenario. The median of the distribution is actually 0 packets lost. In fact, 458 out of 500 scenarios simulated lost *no* packets whatsoever. Out of the minimum 22,500,000 packets admitted into the network, TRUMP/RBCC lost only *551* packets, a loss of roughly 0.002%

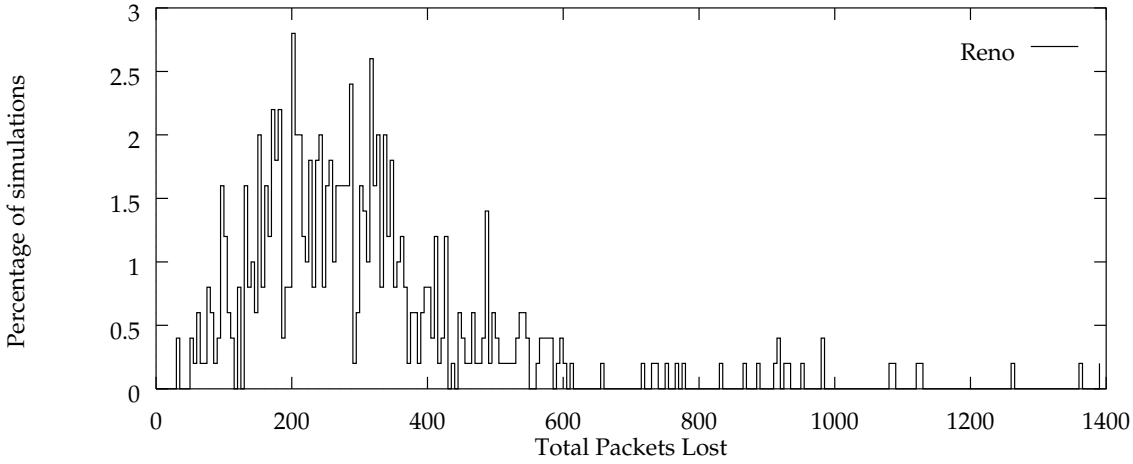


Figure 56: Total Packets Lost by TCP Reno

TCP Reno, on the other hand, shows a very disappointing distribution of packet loss. In fact, all of the 500 simulated scenarios lose packets with TCP Reno. The median of the distribution is 274 packets lost, with 90% of the distribution in the range 75:547 packets lost. TCP Reno loses 155,833 packets in total over the 500 scenarios. Clearly, TRUMP/RBCC provides much better congestion control than TCP Reno, based on packet loss alone.

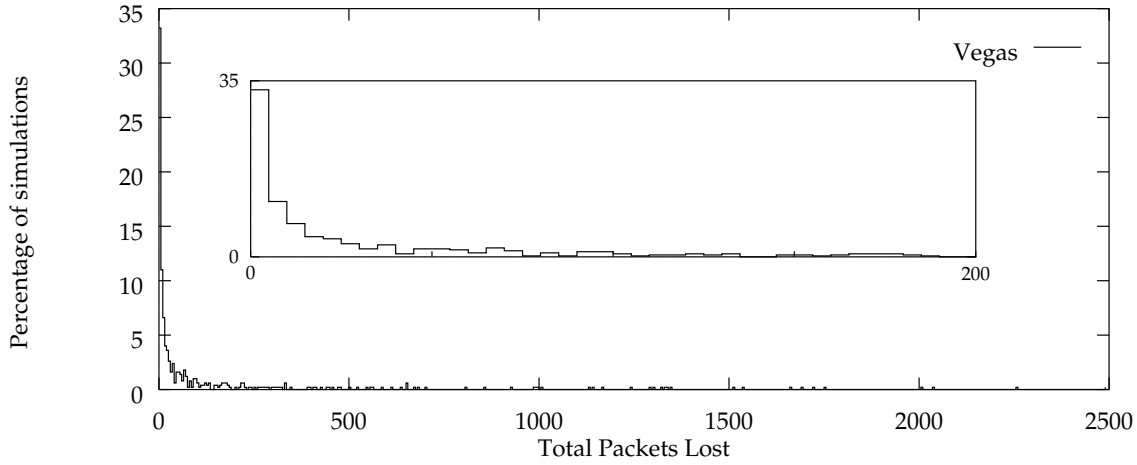


Figure 57: Total Packets Lost by TCP Vegas

TCP Vegas shows better packet loss than TCP Reno, with a median of 14 packets lost and a 90% range of 0:396 packets lost. 97 out of 500 scenarios simulated lost no packets whatsoever. However, in some scenarios, TCP Vegas lost more packets than both TRUMP/RBCC and TCP Reno; the worst per-scenario packet loss was 2,491 packets. The total packet loss over the 500 scenarios was 70,111 packets. Clearly, in some situations, Vegas is very poor at congestion control.

10.7 Highest Average Link Utilisation

In all networks, there is at least one link which should be fully utilised, as it is the bottleneck in the network. A perfect congestion control scheme will ensure that this link *is* fully utilised, so as to ensure the best throughput for the sources limited by the bottleneck.

The distribution of highest average link utilisation per simulation is shown in Figure 58. Clearly, TRUMP/RBCC keeps the bottleneck link at a high utilisation. The distribution has a median of 0.93 with 90% of the distribution in the range 0.84:0.99[2].

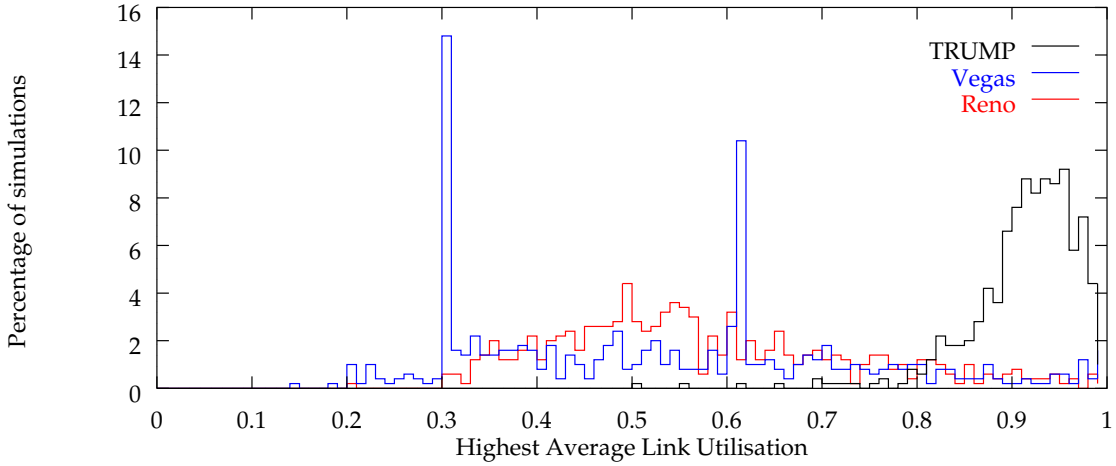


Figure 58: Highest Average Link Utilisation

TCP Reno poorly utilises the bottleneck link. Its distribution has a median of 0.55 with 90%

[2]Normalised utilisation: a utilisation of 1.00 is full utilisation.

of the distribution in the range 0.33:0.82. Thus, sources using Reno achieve lower throughput than with TRUMP/RBCC. Similarly, TCP Vegas show poor bottleneck utilisation with a median of 0.53 and a 90% range of 0.26:0.87, which is close to Reno's distribution. I have no explanation for the two large spikes in the Vegas distribution at 30% and 61% utilisation.

10.8 Effective Bit Rate

Related to the utilisation of the network is the effective bit rate of each source: the total number of bits to be transmitted from a source application to a destination application, divided by the time taken to achieve this.

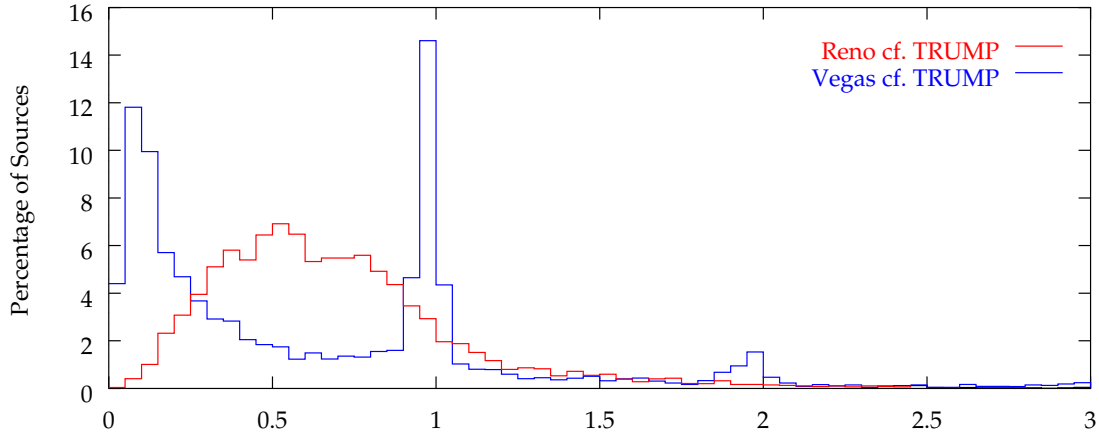


Figure 59: Effective Bit Rate Ratio for All Sources

Figure 59 gives a histogram of the effective bit rate of TCP Vegas sources compared to TRUMP sources (blue plot), and the effective bit rate of TCP Reno sources compared to TRUMP sources (red plot). The graph's x-axis is a ratio: at $x = 0.5$, a Vegas or Reno source achieved half the effective bit rate of the corresponding TRUMP source.

Clearly, TCP Reno generally achieves a lower effective bit rate than TRUMP, for all sources in the 500 random scenarios: only 15% of the Reno sources achieved a better effective bit rate than corresponding TRUMP sources. Given TRUMP's good link utilisation and low packet loss results, this affirms that TCP Reno underutilises network resources.

A large proportion of TCP Vegas sources achieved an effective bit rate equal to the corresponding TRUMP sources: 19% of the Vegas sources achieved a better effective bit rate than corresponding TRUMP sources. However, a significant proportion of TCP Vegas sources had worse effective bit rates than the TCP Reno sources. This result should be viewed in conjunction with Figure 60 below.

In this figure, only the ratios for the 'slowest' TRUMP sources were used: the slowest TRUMP source achieved the lowest effective bit rate in each scenario.

TCP Reno sources exhibit a similar distribution of effective bit rates to Figure 59. TCP Vegas sources, though, achieve much lower effective bit rates than the slowest TRUMP sources. Taken with the Vegas result in Figure 59, this implies that some Vegas sources are being penalised, and have received an unfair allocation of bandwidth. In other words, Vegas' congestion control mechanism is unfair to some sources. This unfairness was observed in Scenario 5 in Chapter 9.

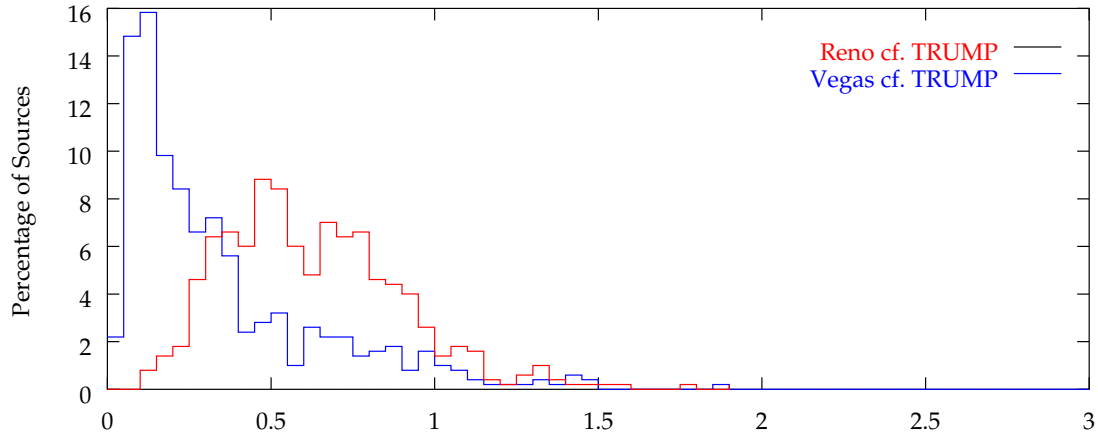


Figure 60: Effective Bit Rate Ratio for 'Slowest' TRUMP Sources

10.9 End-to-end Variance

Although not important to network congestion, the end-to-end variance of sources is important to time-sensitive network traffic such as voice and video data transmission. Because the rate-based congestion control framework admits data uniformly into the network, end-to-end variance in the framework should be lower than for a window-based system such as TCP.

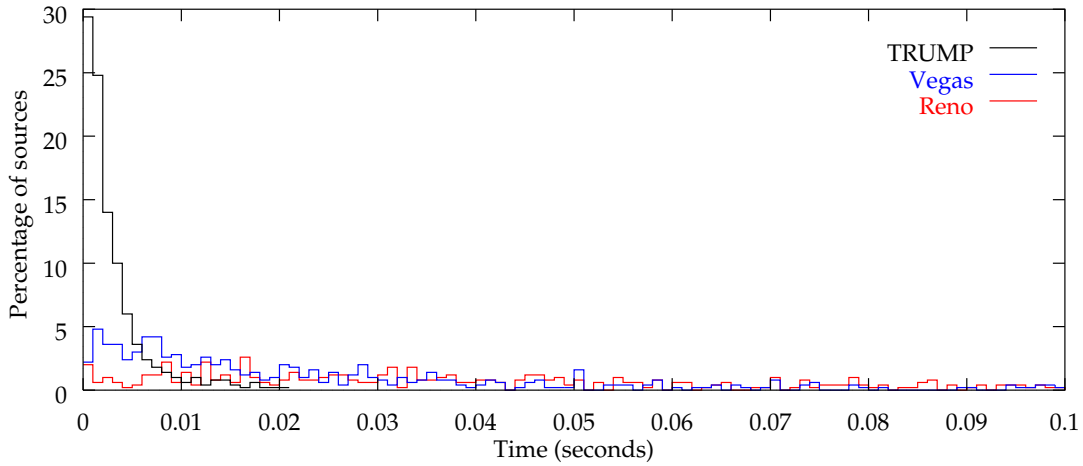


Figure 61: End-to-end Standard Deviation

The end-to-end standard deviation distribution for all sources is shown in Figure 61. All TRUMP sources achieved an end-to-end standard deviation of less than 220 milliseconds, with 90% of the sources obtained a standard deviation less than 6 milliseconds.

90% of TCP Reno sources obtained a standard deviation less than 280 milliseconds, and 90% of TCP Vegas sources obtained a standard deviation less than 145 milliseconds. Clearly, the arrival times of TRUMP packets are more predictable than for TCP.

10.10 Inter-packet Spacing

One of the requirements set down for the rate-based congestion control framework is that packet spacing should be preserved across the network. In the simulations presented, no work has

been done to preserve this by the RBCC routers. Figure 62 shows the change in packet spacing from source to destination, for TRUMP, TCP Reno and TCP Vegas. The figure is a cumulative histogram; the x-axis gives a particular inter-packet spacing change in microseconds, and the y-axis shows the percentage of packet pairs with this change (or less) across the 500 random scenarios. For TRUMP/RBCC, there are 7.7 million inter-packet spacing changes; Reno and Vegas have more due to data retransmissions.

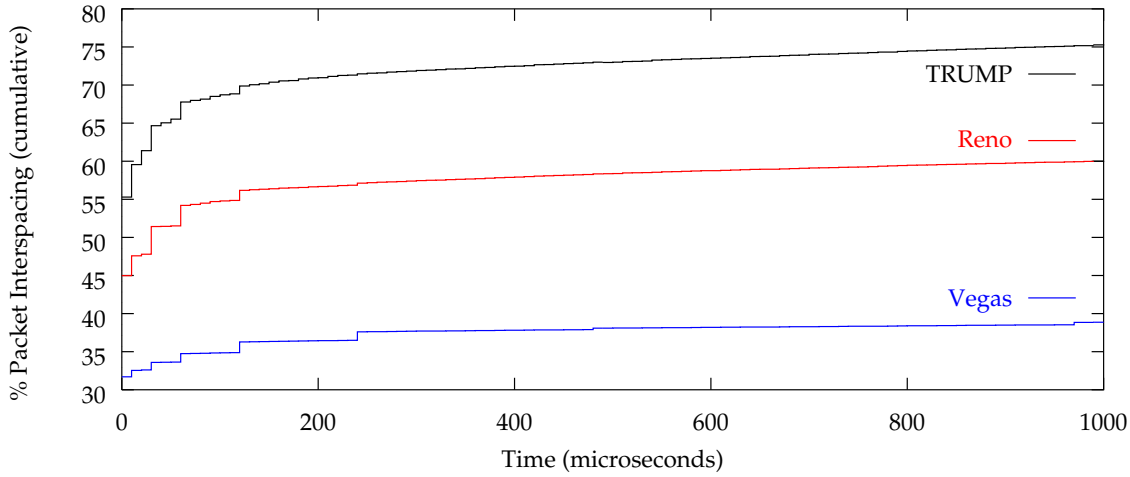


Figure 62: Cumulative Interpacket Spacing Histogram

The figure shows that TRUMP/RBCC preserves inter-packet spacings well, with no effort on its part. 53% of all TRUMP/RBCC packet pairs have no measurable spacing change[3]. This compares quite well to Reno (45%) and Vegas (32%). 87% of TRUMP/RBCC packet pairs have a spacing change less than 1 millisecond, compared to Reno's 60% and Vegas' 39%.

The good spacing results for TRUMP/RBCC are due to the lower router queue lengths: this causes fewer packet delay variations due to queueing, and so spacing is preserved. Although it is desirable to further improve TRUMP/RBCC's inter-packet spacing results, they are clearly superior to TCP's, and should be quite acceptable to real-time data communications.

10.11 ABT Caching Results

In Section 7.5, it was suggested that caching of congestion fields in packets should help to reduce the number of RBCC updates. Section 8.4 described the implementation of field caching in the REAL 4.0 version of RBCC.

Figure 63 shows the distribution of cache misses in the 500 random scenarios. Nearly all of the scenarios simulated kept their cache misses below 1%, and thus their cache hits above 99%. Caching appears to be an effective way of reducing the computational load of RBCC on the network's routers.

After caching has removed most potential ABT updates, there is little ABT work left as router overhead. The median for actual ABT updates is 214 updates per scenario (the total for all 9 routers), with the 90% range between 44:402 updates per scenario. The total number of ABT update operations for all 500 scenarios is 114,876. This is roughly 25 update operations per router for each scenario, a small amount of extra work per router.

[3]The time resolution in REAL 4.0 is 1 microsecond.

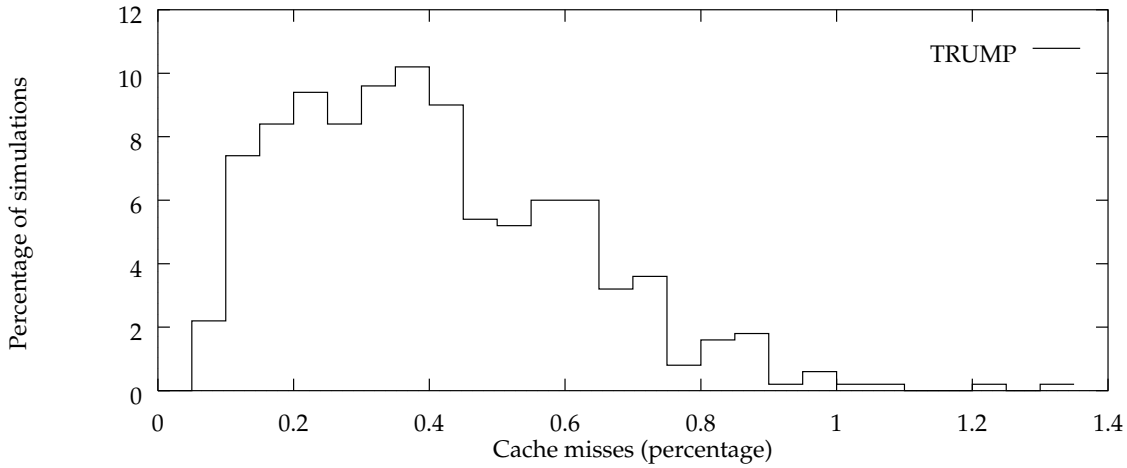


Figure 63: Percentage of Cache Misses

10.12 Summary of Results

In order to confirm that the simulation results of TRUMP/RBCC on a set of hand-picked network scenarios were not abnormal, TRUMP/RBCC was simulated on 500 pseudo-randomly generated network scenarios. As well, TCP Reno and Vegas were simulated on these scenarios to allow for comparisons of simulation results.

For each simulation, several key results were abstracted: this includes average router queue lengths, packets lost, link utilisation, throughput of slowest source and end-to-end variance. These were presented as a set of distributions for the 500 simulated scenarios.

TCP Reno was shown to be a poor network congestion scheme. Router queue lengths were very high, and packets were lost in all scenarios. In most scenarios, the packet loss was high. Link utilisation was very poor for the most congested link, and the sources generally achieved a lower effective bit rate than with TRUMP/RBCC. Finally, end-to-end variance and inter-packet spacing was high, which makes TCP Reno unsuitable for time-sensitive network traffic.

The congestion control of TCP Vegas is much better than TCP Reno, as packet loss is very much improved. However, in many respects, the performance of Vegas is similar to Reno: high router queue lengths, poor link utilisation and poor end-to-end variance. One serious drawback with TCP Vegas is its apparent unfair treatment of long path traffic flows with regards to available bandwidth: this problem alone should rule out the replacement of TCP Reno with TCP Vegas in the current Internet.

The results show that TRUMP/RBCC provides a very good congestion control scheme indeed. Router queue lengths were close to the ideal value of 1, even for severely congested routers. In over 90% of the scenarios simulated, no packets were lost, and very few packets were lost in other scenarios. Link utilisation for the most congested link was close to full utilisation. End-to-end variance and inter-packet spacing changes were very suitable for time-sensitive network traffic. Finally, with implementation techniques such as congestion field caching in routers, the overhead to implement the rate-based congestion control framework using TRUMP/RBCC appears to be low.

Chapter 11

Effect of Parameters on Framework Performance

Recall that the proposed rate-based congestion control framework (see Figure 4) is composed of a number of functions that intercommunicate via several congestion control fields in every network packet, as was shown in Figure 5. Any specific function that meets the framework's requirements can be used as one of the framework's functions. Thus there could be any number of transport protocols, sustainable rate measurement functions, and so on.

In this thesis I have described several specific functions to be used in the congestion control framework. Several of these functions have parameters which affect their operation. In this chapter we will explore how the possible values of these parameters affect the functions and also the overall performance of the framework where these functions are used.

11.1 Available Parameters

The two major function instantiations which have been described are TRUMP, a rate-based Transport protocol, and RBCC, a Sustainable Rate Measurement function. The implementations of RBCC and TRUMP have parameters which can be altered to change their behaviour. RBCC has the following parameters:

- To perform ABT updates based on the two reverse congestion control fields in every packet. This was described in Section 8.4. This parameter can have two values: yes or no.
- To send Rate Quench packets as a form of congestion avoidance. This was described in Section 8.4. This parameter can have two values: yes or no.
- To scale an output interface's available bandwidth when the interface's buffered packet queue length exceeds a threshold of T packets. This was described in Section 4.11 as a form of congestion avoidance. This threshold can have integral values ranging from 1 to MP, the maximum number of packets that the interface can hold.
- The bandwidth scaling value, α, which is used when the old value T is exceeded. Values for this parameter range from 0.0 (exclusive) to 1.0 (inclusive).

The TRUMP transport protocol has one parameter:

- The maximum number of data segments which can be selectively acknowledged in a TRUMP acknowledgment packet. This was described in Section 5.3.1. This parameter can have integral values ranging from 1 to 16.

The set of possible parameter values form a 5-dimensional space with an infinite number of 5-tuples. We will explore some of the 5-space by keeping 4 parameters constant and varying a fifth[1].

11.2 Measurements Made

Most of abstracted measurements presented in this chapter are the same as for the previous chapter where TRUMP/RBCC, TCP Reno and TCP Vegas were compared in the 500 pseudo-random scenarios. The abbreviations given below will be used in the table presented in this chapter.

hiqueue The average buffer queue length (in packets) for the router with the highest average length.

avqueue The average buffer queue length (in packets) for all the routers in the simulated scenario.

hiutil The average utilisation (normalised) for the link with the highest utilisation.

e2edev The end-to-end standard deviation for all sources in the simulated scenario.

hits The total number of cached congestion 'hits' in the simulated scenario, which allowed ABT update decisions to be skipped.

misses The total number of cached congestion 'misses' in the simulated scenario, which forced ABT update decisions to be performed.

abt The total number of ABT table updates for all the routers in the scenario.

lost The total number of packets lost in the simulated scenario.

For each measurement, the median value and the 90% range will be given. The only exception to this is the **lost** result. The median value is not given, as this is nearly always 0. Instead, the total number of packets lost will be given.

11.3 Effect of Rate Quench Packets

Rate Quench packets were described in Section 8.4: If a data packet causes a flow's rate to be lowered in a router's ABT, a Rate Quench packet is returned to the source with the **Return Rate** field set to the flow's new rate in the ABT.

This congestion avoidance mechanism causes traffic flows to throttle back their transmission faster than the usual mechanism of obtaining the new rate via acknowledgment packets, which

[1] A run of 500 scenarios for a particular 5-tuple typically takes between 7 and 10 hours to complete. It is infeasible to thoroughly explore the 5-space.

takes one round trip. The overall effect should be to lower network congestion, as traffic flows are transmitting data at an incorrect rate for less time. However, the Rate Quench packets form a traffic flow themselves, and as a network becomes more congested, more Rate Quench packets are generated, which may lead to even more congestion.

The effect of Rate Quench packets were examined in the 500 pseudo-random scenarios described in the previous chapter. The following table shows the measurements for these scenarios where Rate Quench packets were generated/not generated, and where the remaining four parameters have the values [selacksize=1, revupdates=no, thresh=5, alpha=0.75].

Measurement	No Quenches		Quench Packets	
	Median	90% Range	Median	90% Range
hiqueue	1.686	1.122:2.130	1.677	1.121:2.104
avqueue	0.784	0.434:1.122	0.789	0.437:1.127
hiutil	0.924	0.837:0.987	0.924	0.845:0.995
avutil	0.097	0.061:0.132	0.098	0.060:0.131
e2edev	0.001	0.000:0.006	0.001	0.000:0.006
hits	109052	93060:128708	109052	93060:128685
misses	394	58:817	394	79:826
abt	214	44:402	214	44:401
lost	551	0:0	411	0:0

It is very difficult to distinguish between the two parameter values. Highest queue lengths are smaller, but average queue lengths are slightly higher. Utilisation is negligibly higher, and all other results are equal. We must look at the total number of packets lost to see any significant difference. With no quenches, 551 packets are lost over the 500 scenarios. With Rate Quench packets used, only 411 packets are lost.

The use of Rate Quench appears to lower overall packet loss without affecting other network characteristics such as utilisation or end-to-end variance. Given the results above, I would recommend that Rate Quench packets be used in any implementation of the rate-based congestion control framework.

11.4 Effect of Reverse ABT Updates

A reverse ABT update occurs when the router updates its ABT table when a packet's **Return_Rate** is less than the flow's rate in the table, or when the flow's bottleneck has altered the **Return_Rate**.

Allowing a router to perform reverse ABT updates provides it with 'downstream' information from packets flowing upstream. Without reverse ABT updates, the router only receives downstream congestion information after:

- the packet reaches the flow's source,
- the source alters its transmission rate, and
- a packet with the new transmission rate information from the source reaches the router.

It would appear that reverse ABT updates should help the network infrastructure react more

quickly to overall network congestion. The results below indicate, however, that in fact reverse ABT updates increase network congestion.

The following table shows the measurements for the 500 pseudo-random scenarios where reverse ABT updates were allowed/disallowed, and where the remaining four parameters have the values [selacksize=1, quench=no, thresh=5, alpha=0.75].

Measurement	No Reverse		Reverse Updates	
	Median	90% Range	Median	90% Range
hiqueue	1.686	1.122:2.130	1.737	1.037:2.237
avqueue	0.784	0.434:1.122	0.793	0.394:1.098
hiutil	0.924	0.837:0.987	0.917	0.821:0.984
avutil	0.097	0.061:0.132	0.097	0.056:0.128
e2edev	0.001	0.000:0.006	0.001	0.000:0.008
hits	109052	93060:128708	108893	92861:128302
misses	394	58:817	459	55:1095
abt	214	44:402	290	47:655
lost	551	0:0	550	0:0

Reverse updates have no significant effect end-to-end standard deviation. However, reverse updates cause queue sizes for the most congested router, and for all routers, to be larger. Utilisation across the most-used links decreases, and there are more router ABT updates as expected. Overall packet loss is the same.

At first, the result appears to be counter-intuitive: the reverse congestion information should help routers to determine better rates for traffic flows. This should lead to better network utilisation. Instead, there is higher queue lengths and less utilisation. A hypothesis for this follows. Reverse ABT updates cause more router table changes, which cause more rate changes in the sources. Source rates fluctuate more, and so the network is slightly more unstable and higher congestion results.

Given the result, I would recommend that reverse ABT updates not be used in RBCC. This reduces the overhead of the RBCC on network routers, and keeps router queue lengths slightly lower, with no detrimental effect on network performance.

11.5 Effect of the Threshold, T

One of the framework's design goals, given in Section 4.2, is that 'A congestion control scheme should strive to keep queue lengths at length 1'. The threshold T causes each router to scale the bandwidth of an output interface by α if there are more than T packets queued for transmission on that interface. This is a form of congestion avoidance, as the scaled bandwidth lowers traffic flows' rates through the interface, which will help bring the interface's queue length back to length 1.

If T is small, than the scaling occurs before the queue size is large. This will help to maintain the queue length near 1, but will cause more router ABT updates, and will also cause more traffic flow rate fluctuations, as the queue length will cross the threshold very often.

If T is large (close to the maximum number of packets which can be queued on the interface), then the threshold is rarely crossed, no congestion avoidance is performed, and high queue lengths are permitted.

The effect of threshold values of 2, 3, 4, 5, 7 and 10 packets were examined in the 500 pseudo-random scenarios, where an interface is able to queue up to 15,000 octets (10 data packets). The remaining four parameters have the values [selacksize=1, quench=no, revupdates=no, alpha=0.75]. The following tables give the median and 90% range results for each threshold value.

Median	T=1	T=2	T=3	T=5	T=7	T=10
hiqueue	1.575	1.635	1.660	1.686	1.714	1.712
avqueue	0.779	0.778	0.791	0.784	0.791	0.791
hiutil	0.906	0.920	0.924	0.924	0.924	0.924
avutil	0.096	0.097	0.097	0.097	0.097	0.097
e2edev	0.001	0.001	0.001	0.001	0.001	0.001
hits	108399	108855	108920	109052	109052	108920
misses	677	427	396	394	392	392
abt	512	249	221	214	212	211
lost	522	535	543	551	560	761

90% Range	T=1	T=2	T=3
hiqueue	1.069:1.884	1.152:2.024	1.094:2.013
avqueue	0.429:1.090	0.438:1.129	0.434:1.118
hiutil	0.815:0.983	0.837:0.990	0.835:0.987
avutil	0.054:0.126	0.060:0.131	0.060:0.131
e2edev	0.000:0.008	0.000:0.007	0.000:0.007
hits	92186:126924	93060:128252	92908:128475
misses	52:2586	52:1192	52:844
abt	33:1695	33:688	33:424
lost	0:0	0:0	0:0

90% Range	T=5	T=7	T=10
hiqueue	1.122:2.130	1.017:2.078	1.000:2.140
avqueue	0.434:1.122	0.437:1.131	0.432:1.140
hiutil	0.837:0.987	0.837:0.987	0.837:0.987
avutil	0.061:0.132	0.060:0.131	0.060:0.131
e2edev	0.000:0.006	0.000:0.007	0.000:0.009
hits	93060:128708	92918:128574	92899:128566
misses	58:817	79:815	83:815
abt	44:402	39:393	36:393
lost	0:0	0:0	0:2

From the tables, it is apparent that all of the measurements, except the number of congestion cache misses and ABT updates, increases as T increases. Router queue lengths increase as the threshold is raised. As a byproduct of this, utilisation in the congested router increases as there are more packets ready for transmission. Because of the higher router queue lengths, end-to-end delays increase, and the higher occupancy in queue lengths causes higher end-to-end variance. More packets are lost as higher queue lengths are tolerated. However, because the threshold is crossed less at higher values of T, the number of ABT updates decreases.

The tables show that a lower threshold value provides better congestion control in the rate-based framework. The only drawbacks are a higher number of ABT updates, and a slightly lower network utilisation. If this form of congestion avoidance was to be implemented, I would recommend a value of T between 3 and 6 packets, inclusive.

11.6 Effect of the Scaling Factor, α

Hand-in-hand with the threshold value T is the scaling factor α. When the threshold is crossed, an interface's total bandwidth is scaled by α and its ABT is recalculated. α can take values between 0.0 (exclusive) and 1.0 (inclusive). Values of α near 1.0 have little scaling effect, and should provide little congestion avoidance. Values near 0.0 clamp traffic flows much below their ideal rate (until short-term congestion is reduced), and should have a negative effect on the overall network utilisation.

The effect of α values of 0.125, 0.25, 0.375, 0.5, 0.625, 0.75 and 0.875 were examined in the 500 pseudo-random scenarios. The remaining four parameters have the values [selacksize=1, quench=no, revupdates=no, thresh=5]. The following two tables give the median and 90% range results for each α value.

Median	**α=0.125**	**α=0.25**	**α=0.375**	**α=0.5**	**α=0.625**	**α=0.75**	**α=0.875**
hiqueue	1.692	1.686	1.675	1.682	1.689	1.686	1.691
avqueue	0.792	0.785	0.778	0.787	0.791	0.784	0.779
hiutil	0.914	0.919	0.923	0.923	0.924	0.924	0.925
avutil	0.097	0.097	0.097	0.097	0.097	0.097	0.097
e2edev	0.001	0.001	0.001	0.001	0.001	0.001	0.001
hits	107293	108518	108920	109052	109052	109052	109040
misses	394	394	393	393	394	394	394
abt	219	214	214	214	214	214	214
lost	27389	4130	357	408	473	551	639

90% Range	**α=0.125**	**α=0.25**	**α=0.375**	**α=0.5**
hiqueue	1.000:2.460	1.017:2.119	1.017:2.012	1.020:2.026
avqueue	0.432:1.216	0.432:1.153	0.437:1.141	0.434:1.131
hiutil	0.784:0.995	0.824:0.987	0.835:0.987	0.842:0.991
avutil	0.055:0.132	0.060:0.132	0.060:0.131	0.056:0.127
e2edev	0.000:0.013	0.000:0.006	0.000:0.006	0.000:0.006
hits	81070:133242	92699:129335	93060:128841	92896:128517
misses	79:928	55:833	79:837	83:823
abt	33:468	33:409	44:409	39:400
lost	0:4	0:0	0:0	0:0

90% Range	**α=0.625**	**α=0.75**	**α=0.875**
hiqueue	1.020:2.071	1.122:2.130	1.120:2.161
avqueue	0.434:1.138	0.434:1.122	0.432:1.121
hiutil	0.842:0.991	0.837:0.987	0.845:0.993
avutil	0.056:0.127	0.061:0.132	0.056:0.127
e2edev	0.000:0.006	0.000:0.006	0.000:0.007
hits	92893:128498	93060:128708	92901:128520
misses	83:823	58:817	79:831
abt	39:398	44:402	33:393
lost	0:0	0:0	0:0

Low values of α lower the utilisation of the highest utilised link as predicted. Router queue

lengths increase and packet loss also increases as α approaches 1.0. The number of ABT updates is not substantially affected by any value of α.

However, as α approaches 0.0, router queue lengths increase and packet loss increases dramatically. From the results given in the tables above, I would recommend α values in the range 0.375 to 0.5, where the T/α congestion avoidance scheme is implemented.

11.7 Effect of the Selective Acknowledgment Size

Another design goal for the rate-based congestion control framework, given in Section 4.2, is that retransmission schemes such as Selective Acknowledgment should be used, as they retransmit packets only when necessary. This gives a parameter in the framework: the number of data packets that a Selective Acknowledgment can acknowledge.

A higher number of data packets acknowledged per ack packet lowers the required rate for the acknowledgment traffic flow, and thus lowers the load on the network. However, the drawback of a higher number of data packets acknowledged per ack packet is that the framework's congestion information takes longer to reach the source, as the information is delayed until a 'full' selective acknowledgment packet is transmitted. Another drawback to Selective Acknowledgments is the complexity that they add to a transport protocol.

The TRUMP protocol, implemented in REAL, allows between 1 and 16 data packets to be acknowledged in each acknowledgment packet. The effect of selective acknowledgment sizes 1, 2, 4, 8 and 16 data packets were examined in the 500 pseudo-random scenarios. The remaining four parameters have the values [quench=no, revupdates=no, thresh=5, alpha=0.75]. The following two tables give the median and 90% range results for each selective acknowledgment size.

Median	Size 1	Size 2	Size 4	Size 8	Size 16
hiqueue	1.686	1.432	1.285	1.343	1.649
avqueue	0.784	0.743	0.715	0.716	0.726
hiutil	0.924	0.925	0.923	0.924	0.925
avutil	0.097	0.097	0.097	0.098	0.100
e2edev	0.001	0.001	0.001	0.002	0.004
hits	109052	81726	67469	60379	56829
misses	394	427	504	518	471
abt	214	222	252	268	258
lost	551	582	617	710	835

90% Range	Size 1	Size 2	Size 4	Size 8	Size 16
hiqueue	1.122:2.130	1.000:1.827	1.000:2.133	1.000:2.365	1.000:2.331
avqueue	0.434:1.122	0.421:1.060	0.405:1.007	0.404:1.052	0.411:1.069
hiutil	0.837:0.987	0.843:0.990	0.843:0.993	0.842:0.990	0.841:0.990
avutil	0.061:0.132	0.056:0.125	0.057:0.124	0.063:0.127	0.067:0.131
e2edev	0:0.006	0:0.007	0:0.012	0:0.031	0:0.031
hits	93060:128708	70437:97142	57886:80942	49765:69982	48604:66874
misses	58:817	106:866	142:988	145:955	157:831
abt	44:402	37:407	36:471	35:483	46:475
lost	0:0	0:0	0:0	0:1	0:3

The tables show that higher selective acknowledgment size increase packet loss, queue lengths and ABT updates. End-to-end variance also increases, and this is due to the extra delays in acknowledging several data packets. Surprisingly, a selective acknowledgment size of one appears slightly worse than a size of two: router queue sizes are increased, and network utilisation is lowered across the most congested link. However, the best packet loss results are obtained for a selective acknowledgment size of one.

The effects on other measurements are not significant. Given that the results indicate some problems with high selective acknowledgment sizes on queue lengths, packet loss, end-to-end variance and most congested link utilisation, I would suggest that small selective acknowledgment sizes be used. Taking into account the packet loss results and the complexity of implementing a Selective Acknowledgment scheme, I would recommend that selective acknowledgments *not* be used in the rate-based congestion control scheme, and that transport protocols implement a 'One data packet, one acknowledgment packet' Acknowledgment scheme.

Rate Quenches also help to mitigate the effect of network delays, as was shown in Scenario 9 in Chapter 9. As large selective acknowledgments add delays to the network, then I would also highly recommend the use of Rate Quenches where large selective acknowledgment sizes are employed.

11.8 Effect of the Packet Dropping Function

If there is no room to queue a newly-arrived packet in a router, one or more packets must be dropped by the router. The function which determines which packet(s) are dropped is the Packet Dropping Function. The following dropping functions are available in the REAL network simulator:

Drop Tail: The newly-arrived packet is dropped.

Drop Head: The oldest queued packet in the router is dropped.

Drop Random: An already-queued packet in the router is randomly chosen and dropped.

Decongest First: The traffic flow with the most bytes queued in the router is found, and the most recently-arrived packet for the flow is dropped.

Decongest Last: The traffic flow with the most bytes queued in the router is found, and the oldest queued packet for the flow is dropped.

In fact, more than one packet may be dropped in the last four schemes so that enough room is made available to queue the newly-arrived packet. For example, several small acknowledgment packets may need to be dropped to queue a newly-arrived data packet.

In order to distinguish between the results of the five schemes, each was run on those randomly-generated scenarios where packets were lost with TRUMP/RBCC. Other parameter values were set at [selacksize=1, revupdates=no, quench=no, thresh=5, alpha=0.75].

Median	Drop Tail	Drop Head	Drop Random	Decongest First	Decongest Last
hiqueue	1.815	1.844	1.947	1.762	1.827
avqueue	0.929	0.891	0.925	0.907	0.909
hiutil	0.921	0.923	0.923	0.923	0.923
avutil	0.098	0.098	0.098	0.098	0.098
e2edev	0.007	0.006	0.007	0.007	0.008
hits	113270	113268	113261	113526	113515
misses	684	688	692	692	696
abt	308	311	308	310	320
lost	551	640	614	551	549

90% Range	Drop Tail	Drop Head	Drop Random	Decongest First	Decongest Last
hiqueue	1.451:2.071	1.438:2.177	1.451:2.207	1.496:2.136	1.370:2.158
avqueue	0.702:1.089	0.711:1.207	0.721:1.230	0.703:1.184	0.689:1.167
hiutil	0.823:0.971	0.820:0.964	0.819:0.964	0.847:0.989	0.820:0.964
avutil	0.053:0.114	0.054:0.115	0.054:0.115	0.054:0.114	0.054:0.114
e2edev	0.001:0.017	0.001:0.014	0.001:0.015	0.001:0.015	0.001:0.015
hits	99503:129323	99503:129320	99497:129320	99503:129311	99503:129320
misses	297:1081	297:1091	303:1118	279:1086	385:1208
abt	136:497	136:495	136:497	136:495	136:495
lost	1:32	1:40	1:32	1:32	1:32

The two schemes with the best combined packet loss/queue length results are Decongest First and Drop Tail. Decongest First has better queue lengths, network utilisation and end-to-end variance than Drop Tail. Despite this, I would argue for the use of Drop Tail as the preferred packet loss mechanism, as it is simple and imposes less overhead on routers than Decongest First.

11.9 Conclusion

The parameters available within the implemented functions of the rate-based congestion control framework do influence the characteristics of the framework, congestion and otherwise. At least two of the five parameters described above, the threshold T and the scaling factor α, have a significant negative impact on the framework's congestion control operation for certain values. Other parameters have some impact, but in general the framework works quite well for all their values.

If the congestion control framework was deployed with the function instantiations examined in this chapter, I would recommend the following parameter values:

- Rate Quench packets should be used,
- No reverse ABT updates should be performed,
- A threshold value of between 3 and 6 packets inclusive should be used,
- An α value of between 0.375 and 0.5 inclusive should be used, and

- No selective acknowledgments should be used. Instead, 'One data packet, one acknowledgment packet' should be used.
- Drop Tail should be used as the packet dropping mechanism.

11.10 Results with Recommended Parameters

A 5-tuple of parameters was chosen from the range of recommended parameters above: Rate Quench packets, no reverse updates, Drop Tail as the packet dropping mechanism, a threshold value of 4 packets, and an α value of 0.4. The following table gives the results in the 500 random scenarios, for these parameters against the 5-tuple used in Chapter 10.

Measurement	**Ch. 10 Parameters**		**Recommended Parameters**	
	Median	**90% Range**	**Median**	**90% Range**
hiqueue	1.686	1.122:2.130	1.647	1.122:2.058
avqueue	0.784	0.434:1.122	0.784	0.437:1.130
hiutil	0.924	0.837:0.987	0.922	0.837:0.990
avutil	0.097	0.061:0.132	0.097	0.060:0.131
e2edev	0.001	0.000:0.006	0.001	0.000:0.005
hits	109052	93060:128708	108920	92699:128565
misses	394	58:817	403	83:928
abt	214	44:402	221	35:441
lost	551	0:0	259	0:0

The recommended parameter values give a 47% drop in packet loss, with a lowering of the queue average in the most congested routers. Link utilisation for the most congested link is slightly lower, as is the end-to-end variance. With the threshold T lower, congestion avoidance occurs more frequently, and so the number of ABT updates has increased. The result is a small increase in the amount of RBCC work performed which gives a marked improvement in packet loss, offset by a slight lowering in link utilisation.

By tuning the parameters available with TRUMP and RBCC, the rate-based congestion control framework achieves 600 times fewer packets losses than TCP Reno, and 270 times fewer packets losses than TCP Vegas, in the 500 randomly-generated scenarios. Overall, throughput and network utilisation is improved, end-to-end standard deviation is lower, and the extra workload on the intermediate routers appears small. The framework provides excellent congestion control for connectionless packet-switched networks.

Chapter 12

Conclusion

12.1 Introduction

This thesis has described the design and testing of a rate-based framework for controlling network congestion in connectionless packet-switched networks.

Network congestion occurs when the demands on the resources of a network are too great; typically this results in increasing queue sizes in routers, causing packet delays and eventually packet loss. In the networks under consideration, congestion control attempts to partition network resources and control the sources of network traffic so that the burden placed on the network is sustainable and does not cause congestion.

Traditional congestion control mechanisms that are end-to-end based (such as TCP) have typically relied on indirect evidence to determine if, and how, a network is congested: loss of packets and changes in round-trip times. These mechanisms have been shown to be poor both at controlling congestion, and at partitioning network resources fairly.

Other techniques such as packet queueing and packet loss schemes have been suggested to alleviate some of the problems with the end-to-end schemes. However, these can only be employed when portions of the network are already congested.

Newer congestion control mechanisms such as DecBit and Source Quench have been proposed which feedback explicit information about network congestion to the source. As they are binary-feedback schemes, they take several round-trip iterations to bring each source to a transmission level which gives peak network power.

Alternatives to these traditional forms of congestion control are techniques which are rate-based. Given that oversubscribed network resources are typically links with a fixed bit rate, it should make sense to deliver a value for a sustainable transmission rate to each source. Transmitting at this sustainable rate, a source should not congest the network. Such explicit rate feedback schemes as EPRCA have been adopted in connection-style networks such as ATM. The application of such techniques in connectionless packet-switched networks might overcome some of the drawbacks of traditional control schemes. However, this would require adaptation of the techniques to overcome the substantial differences in network architecture.

12.2 A New Rate-Based Control Framework

Given the problems of traditional congestion control mechanisms and the promise of rate-based techniques, I proposed a rate-based congestion control framework for connectionless packet-switched networks which should remedy these problems. The design considerations for the framework, given in Chapter 4 were that:

- Transport congestion control should use congestion information from routers,
- Dropping packets is highly undesirable,
- Router queue sizes >1 are highly undesirable,
- Use of round-trip timers is undesirable,
- Use of sliding windows for flow and congestion control is undesirable,
- Packet retransmission should be done only if necessary,
- Separation of error control and flow/congestion control is desirable, and
- Packet admission and readmission should be smooth.

The resulting framework consists of a rate-based Transport Layer whose rate is controlled by congestion information obtained from all routers along a traffic flow's path. The routers employ a distributed resource allocation algorithm to determine this rate, known as a *sustainable rate*. To do this, each router must monitor the set of traffic flows which it is routing; specifically, the set of traffic flows on each output interface.

The framework, thus, is a combination of operations performed by the end-host (i.e the Transport and Network Layers) and the intermediate routers (i.e the Network Layer). At the end-host, the two layers intercommunicate congestion information as described in Section 7.4. Within the network proper, all packets carry a number of congestion fields:

Desired_Rate: The bit rate which the source of this traffic desires.

Rate: The bit rate which has been allocated to this flow of traffic by a router.

Bottle_Node: The router which set the value of the **Rate** field in the packet.

Return_Rate: The value of the last **Rate** field which reached the destination.

Other fields indicate to routers when traffic flows terminate.

The framework operates as follows:

1. The Transport Layer uses a rate-based congestion control scheme, using rate information provided by the Network Layer.
2. A source admits packets for each traffic flow into the network uniformly spaced in time.
3. The routers in the Network Layer implement a congestion control scheme, part of which measures the sustainable traffic rate across the network for each traffic flow: this is performed by the Sustainable Rate Measurement function. This sustainable rate measurement is passed to the destination machine and then via acknowledgment packets to the source machine.
4. Routers implement both congestion avoidance and congestion recovery mechanisms as the rest of the Network Layer congestion control scheme. Routers partition the available resources fairly amongst traffic flows, in both uncongested and congested operating regimes.

12.3 Framework Functions

The proposed framework consists of a number of functional elements. Some functions, such as packet dropping mechanisms, can be implemented with existing algorithms. Two elements, that of the Transport Layer function, and that of the Sustainable Rate Measurement function, had to be instantiated.

A transport protocol, TRUMP, which satisfied the requirements for a Transport Layer, was described in Chapter 5. Although this protocol was specifically designed to complete the proposed congestion control framework, it has been specified and verified in enough detail to be considered a 'real' and implementable protocol. TRUMP provides a reliable unidirectional connection from a source application to a destination application across an unreliable network. Features of the protocol include:

- Error detection and recovery using selective acknowledgment and retransmission.
- Explicit and implicit connection setup and connection teardown.
- Rate-based flow and congestion control, using information passed to TRUMP by the Network Layer.
- An infinite 'buffer' size: the source can transmit all of a message before any of the acknowledgments have arrived. This aids high throughput.
- Only one timer required per message during data exchange, which is in the source. The timer is not based on the round-trip time.
- Up to 16 possible types of transport header/data encryption. The encryption covers all fields except for the protocol version number, the encryption type and the segment checksum.

Although TRUMP is a complex protocol, its correctness was partially proven using the Spin protocol verification tool in Appendix B.

An algorithm which meets the requirements of a Sustainable Rate Measurement function, RBCC, was presented in Chapter 6. RBCC is an algorithm distributed across all routers of the network, which calculates a set of sustainable rates for all flow sources to achieve peak network power. The design goals for RBCC were that:

- The function should allocate enough bandwidth to meet each traffic flow's desired rate, if possible.
- The function should allocate bandwidth to flows so that no output interface's available bandwidth is exceeded.
- If a router is a bottleneck, the function should communicate its allocated rates to each source affected, and to other routers.
- The function should allocate bandwidth to flows, taking into account any bottlenecks upstream (towards the source) and downstream (towards the destination).
- Flows should be free to change their desired rate at any time and to any value. The function should cope with changes in a traffic flow's desired rate.

RBCC uses a number of heuristics, to determine if a packet received at a router should cause a recalculation of the set of sustainable rates allocated by the router, held in an Allocated Bandwidth Table (ABT). This recalculation is performed by an algorithm which must meet several criteria:

- Allocation of rates amongst the flows in the ABT must be fair.
- A traffic flow's rate allocation must not exceed the desired rate of its source, nor may it exceed the rate set by a bottleneck router which is not this router.
- A traffic flow's rate allocation cannot be set to zero or to ∞.
- The sum of allocated rates in the ABT must not exceed the output interface's bit rate.

The proposed algorithm is given in Section 6.2.3. Various issues with the framework were raised in Chapter 7: the effectiveness of the framework as a feedback loop; the identification of traffic flows within the network; the required congestion fields within packets; the intercommunication of congestion information between network layers, and the complexity and overhead of the framework itself. Some issues raised still need further examination.

12.4 Experimental Results

The proposed rate-based congestion control framework was tested experimentally using a network simulator, for two main reasons:

1. Theoretical analysis of the framework as a control scheme would limit its validation to very simplistic network architectures, and
2. It was desirable to observe the frameworks control of congestion on a number of realistic network scenarios. Experimental testing would also reveal the sensitivity of the framework's parameters on its congestion control performance.

Experiments described in Chapter 9 concentrated on network scenarios which were hand-crafted to reveal any deficiencies with the control of congestion, and the fairness, in the framework. The framework was also compared with two traditional end-to-end control schemes, TCP Reno and TCP Vegas.

The hand-crafted network experiments showed that the proposed framework appeared to deal with network congestion more than adequately: no packets were lost in any of the experiments, and queue lengths in routers stayed close to one packet queued. Overall, link utilisation was high, and end-to-end variance was low. The framework controlled congestion, and allocated rates to all sources in a fair manner. In comparison, TCP Reno and TCP Vegas achieved poorer congestion control, with large packet loss, high router queue lengths, lower link utilisation and higher end-to-end variance.

In order to give a more statistically valid experimental analysis of the framework, 500 network scenarios were pseudo-randomly generated. Each scenario had 15 traffic flows, 9 routers, and an arbitrary network topology. Link capacities were arranged so as to increase the likelihood of congestion conditions.

The proposed framework was simulated within each of the 500 network scenarios, and the results described in Chapter 10. Again, TCP Reno and TCP Vegas were also simulated to provide a comparison with traditional control techniques. The results confirmed those obtained with the hand-crafted scenarios: the proposed framework controlled congestion very well, and allocated rates to all sources in a fair manner.

The effect of specific parameters within the TRUMP and RBCC components of the framework was examined in Chapter 11. Most of the parameters had little effect on the framework's congestion control performance, except for extremes in parameter values. Two of the five parameters examined, the threshold T and the scaling factor α, had a significant negative impact on the framework's congestion control operation for certain values. A set of parameter values was recommended, and their effect on the framework was shown to have a positive effect on congestion control performance.

12.5 Future Work

During the course of this work, several issues with the proposed congestion control framework have been raised but left unanswered: many issues were discussed in Chapter 7. The constraints on the size, duration and scope of this research have prevented me from exploring these issues. They should provide several areas for future research into rate-based network congestion control.

The suitability of the proposed framework as a congestion control mechanism has only been experimentally verified. Rigorous, mathematical analysis of the framework as a distributed algorithm should help to confirm its worthiness.

Two main functions in the framework are the Transport protocol, TRUMP, and the Sustainable Rate Measurement function, RBCC. TRUMP as a protocol was not specified enough to be fully implemented in a real protocol stack. Its specification should be completed, and an implementation of TRUMP created for a suitable protocol stack. Similarly, RBCC should be implemented as part of a real protocol stack.

Although the thesis has shown RBCC implemented within routers, in a real network it must also be implemented within sources of traffic flows: this arises as typical source machines must multiplex many independent application data flows onto one (or more) network interfaces. RBCC within the source would ensure that the network interfaces' bandwidth is shared fairly amongst the application data flows. An implementation of RBCC would have to be done for sources as well as routers.

A combined TRUMP/RBCC implementation would allow studies of the proposed congestion control framework in real-life network scenarios. This would verify the results of the network simulations given in this thesis, and allow the effects of Link Layers and hardware on the framework to be examined. A real-life implementation of RBCC would facilitate the study of the computational overhead of the the framework on routers, and provide the incentive to improve its implementation efficiency.

Section 5.3.2 describes TRUMP's use of a lazy retransmission strategy to avoid the setting of any round-trip timer. This can cause excess network load when network capacity is high and round-trip delays are large. An alternate method of prompting the destination for acknowledgments should be studied, and its effect upon network load should be compared to lazy retransmission.

The constitution of a flow of traffic within the framework was discussed in Section 7.2, and resolved to be the flow of data between two applications. This does add a burden on the framework in comparison to other alternatives discussed. The feasibility and overhead of the other flow identify alternatives should be examined.

The framework deals with route changes in a primitive fashion. Further studies should be conducted to allow the framework to react more intelligently. A synergy may be achieved if the congestion control framework was coupled with the network's route control framework: this needs to be investigated.

Within RBCC itself, several heuristics were described: the T, α congestion avoidance mechanism and the ABT update decisions. A fruitful area of research would be to examine these heuristics and replace them with ones that improve the congestion control of the framework. The congestion avoidance mechanism is particularly primitive, and deserves special attention.

On a higher level, both TRUMP and RBCC are only particular function instantiations which can be used within the congestion control framework. Other Sustainable Rate Measurement function instantiations may provide better congestion control within the framework than RBCC. Certainly, Transport Layers other than TRUMP are required to provide particular value-added services to particular applications.

The Fair Share rate allocation scheme [Charney 94] should be investigated to determine if it can be used as is, or modified, to work as a Sustainable Rate Measurement function within the proposed framework. If so, its rate allocation performance and overhead should be compared with RBCC.

Communication within the framework components is performed by a number of congestion fields in each packet, and infrequently by the exchange of Rate Quench packets. The number and types of fields can be changed to provide such things as flow priorities and congestion-oriented qualities of service. This should be studied. The trigger for Rate Quench packets, which network components may send them, and their effect upon sources, also deserves further study.

12.6 Summary

The early adoption of sliding windows for flow control in connectionless packet-switched networks precluded the use of rate-based techniques to control congestion. Sliding window flow control was modified to perform end-to-end congestion control, but several problems (round-trip estimation, bursty packet admission, window side-effects) render congestion control with these techniques difficult.

An alternative approach is to use rate-based techniques for congestion control. Such techniques have been adopted for congestion control in other network architectures. This thesis has examined the requirements for a rate-based congestion control framework for connectionless packet-switched networks. From these requirements, a framework has been designed, and the functionality (and design criteria) for its components has been instantiated. Two major components — a rate-based Transport protocol, TRUMP, and a Sustainable Rate Measurement function, RBCC — were described.

In order to verify the effectiveness of the congestion control framework, it was analysed experimentally in a large number of simulated network scenarios. The parameters of the framework's components were analysed, and the framework congestion control performance was compared with traditional Internet control mechanisms.

The rate-based control framework was shown to excel at congestion control, giving a packet loss result orders of magnitude lower than the traditional control mechanisms. Results for router queue lengths, link utilisation, rate allocation and end-to-end variance confirm the congestion control performance of the framework.

The framework has been designed to be modular, and to allow the individual framework components to be modified or replaced, as long as the framework's operational requirements are met. Further research may see improved components, or new components which enhance the effectiveness of the framework's congestion control, or reduce the overhead of the framework on the network's components.

Appendix A

Glossary

This appendix gives a glossary of technical terms used throughout this thesis. Where terms are taken from another source, the source is cited in the definition of the term. Italicised words represent terms that are defined elsewhere in this glossary.

Acknowledgment A response sent by a *Receiver* to indicate successful reception of information. Acknowledgments may be implemented at any level [Comer 88].

ACK Abbreviation for *Acknowledgment* [Comer 88].

Application A computer program.

Bandwidth The difference between the limiting frequencies of a continuous frequency spectrum [Stallings 91]. Used colloquially in this thesis to mean the overall *Bit Rate* capacity of a *Link*.

Binary Feedback Mechanism A mechanism which returns a single bit of data as its feedback. DecBit [Jain *et al* 87] and PRCA [Hluchyj 94] are binary feedback congestion control mechanisms.

Bit Rate The rate at which bits are transmitted, in bits per unit time. See also *Data Rate*.

Bottleneck The network resource which is limiting the *Bit Rate* of a transmission.

Buffer An amount of computer memory used to store information. Routers have a finite amount of buffers (also known as *Buffer Space*) in which to hold incoming *Packets* for forwarding. *Hosts* also have a finite amount of buffer space to hold incoming packets and packets which they have generated but not yet transmitted.

Buffer Occupancy The current level of *Buffer* use, either in terms of *Packets* queued, or bits queued.

Byte A colloquial term for *Octet*.

Capacity The highest *Data Rate* that can be sustained through a network from a *Source* to a *Destination*, or through part of a network. See also *Sustainable Rate*.

Congestion The state when the resource demands on a network (or a part of a network) exceeds its *Capacity* [Jain & Ramakrishnan 88].

Congestion Avoidance A congestion control scheme allows a network to operate in the region of low *Delay* and high *Throughput*, where *Power* is at its optimum
[Jain & Ramakrishnan 88]. See Section 1.3 for more details.

Congestion Control A congestion control scheme tries to bring a network back into an operating state, when demand has already exceeded *Capacity* [Mankin & Ramakrishnan 91].

Connection The path between two *Protocol* modules that provides reliable delivery service [Comer 88].

Connection Setup The stage of a *Connection-Oriented Service* where the *Connection* is established between two *Hosts*.

Connection Teardown The stage of a *Connection-Oriented Service* where the *Connection* between two *Hosts* is broken.

Connectionless Service Characteristic of the *Packet* delivery service offered by most hardware and the Internet Protocol. The connectionless service treats each packet as a separate entity that contains the *Source* and *Destination* address. Usually, connectionless services can drop packets or deliver them out of sequence [Comer 88].

Connection-Oriented Service A communication service which is based upon the concept of a *Connection.*

Data Rate The rate of data in bits per unit time. See also *Bit Rate*.

Datagram The basic unit of information passed by a Network Layer across a network. It contains a *Source* and *Destination* address along with data [Comer 88]. See also *Frame*, *Packet* and *Segment*.

Delay The amount of time it takes for data (or a quantum of data such as a *Packet*) to leave a *Source* and reach its *Destination*. See also *End-to-End Time*.

Destination The *Host* which is the intended destination of a flow of data across a network.

End-to-End Something which is performed by a *Source* and *Destination*, of which the intermediate components of the network have no knowledge. *Flow Control* is an end-to-end mechanism.

End-to-End Time The time between the transmission of a *Packet* at the *Source* and its reception at the *Destination.* See also *Delay*.

Field An individual item of data within a *Header*.

Flow Control Control of the rate at which *Hosts* or *Routers* inject *Packets* into a network. Flow control mechanisms can be implemented at various levels [Comer 88].

Frame Literally, a *packet* as it is transmitted across a serial line [Comer 88]. In this thesis, it refers to a packet as it travels across a *Link* between two peer Link Layers.

Gateway A dedicated computer that is attached to two or more networks and routes *Packets* from one to the other. In particular, an Internet gateway routes IP datagrams among the networks to which it connects [Comer 88]. Although the term has a completely different meaning within the OSI Reference Model, its meaning as defined here is widely used in the Internet community, and is used as such throughout this thesis. See also *Node*, *Router* and *Packet Switch*.

Go Back N Acknowledgment A method of data acknowledgment where the first loss of data is indicated. Upon receipt of this acknowledgment, a *Source* must retransmit all data from the point of first loss onwards. Compare *Selective Acknowledgment* and *Selective Retransmission*.

Header System defined control data that precedes user data [Stallings 91].

Host A computer attached to a network that is not a *Router*. It is usually a *Source* or *Destination* (or both) of data flows.

Interface The physical connection of a *Host* or *Node* to a network. A network-connected computer may have one or more interfaces.

Inter-packet Delay The interval between successive *Packet* transmissions from a *Source* to a particular *Destination*.

Latency The propagation time of a bit or *Frame* across a *Link*.

Leaky Bucket A single-server queueing system with constant service time [Turner 86].

Link A physical connection between two *Nodes* on a network. Data flows between the *Nodes* on a network via links.

Link Layer The communications layer that provides for the reliable transfer of information across a *Link* [Stallings 91].

Load The *Data Rate* being offered to a network, or part of a network. See also *Capacity*.

Network Layer The communications layer that provides for the delivery of information across a number of *Links*.

Node The computers in a network which transfer data between *Links*. In a *Packet-Switched Network*, a *Node* is also known as a *Packet Switch* or *Gateway*. See also *Router*.

Octet A group of eight bits. Often colloquially known as a *Byte*.

Output Interface The *Interface* used by a *Host* or *Node* to transmit on a *Link*.

Packet The basic unit of data flow in a *Packet-Switched Network*. Packets can vary in size between a fixed range of octets. See also *Datagram*, *Frame* and *Segment*.

Packet Admission The insertion of a *Packet* into a network via an *Output Interface*.

Packet Switch A dedicated computer that is attached to two or more networks and routes *Packets* from one to the other. See also *Gateway*, *Node* and *Router*.

Packet Switched Network A network which uses *Packet Switching* as its method of data flow.

Packet Switching A method of transmitting data through a network, in which long messages are subdivided into *Packets*, which are transmitted one at a time across the network via *Packet Switches* [Stallings 91].

Packet Loss The loss of *Packets* at a *Node* due to the lack of available *Buffer* space to queue them.

Path Length The length of the communications path between the *Source* and *Destination*. This may be measured in distance, delay, or by the number of intermediate *Nodes*. See also *End-to-End Time*.

Power The ratio of *Throughput* to *Delay* [Mankin & Ramakrishnan 91].

Protocol A set of rules that govern the operation of functional units to achieve communication [Stallings 91].

Queue A first-in, first-out *Buffer*.

Queue Length An alternative term for *Buffer Occupancy*.

Receiver An alternative term for *Destination*.

Response Time An alternative term for *Delay*.

Round Trip Time The *Delay* for data or a *Packet* to reach a *Destination* and the *Acknowledgment* for that packet to return to the *Source* from the destination.

Route The path that network traffic takes from its *Source* to its *Destination*.

Router A device used to connect two networks that may or may not be similar. The router employs an internet *Protocol* present in each router and each *Host* of the network. The router operates at the Network Layer (layer 3) of the OSI Reference Model [Stallings 91].

Segment The unit of transfer sent from TCP on one machine to TCP on another. Each segment contains part of a stream of bytes being sent between the machines as well as additional fields that identify the current position in the stream and contains a checksum to ensure validity of received data [Comer 88]. See also *Datagram*, *Frame* and *Packet*.

Selective Acknowledgment A method of data acknowledgment where only the data that has been lost is indicated as such. Compare *Go Back N Acknowledgment*.

Selective Retransmission A method of retransmission in the face of network errors where only that data which has been lost is retransmitted.

Sender An alternative term for *Source*.

Sequence Number A *Field* within a *Header* that uniquely identifies the location of a *Packet's* contents within a *Traffic Flow*.

Sliding Window Characteristic of *Protocols* that allow a *Sender* to transmit more than one *Packet* of data before receiving an *Acknowledgment*. After receiving an acknowledgment for the first packet, the sender 'slides' the packet window and sends another. The number of outstanding packets or *Octets* is known as the *Window Size* [Comer 95].

Source The *Host* which is the source of a flow of data across a network to a *Destination*.

Sustainable Rate The maximum *Bit Rate* at which a *Source* can transmit data to a *Destination* without causing congestion in the intermediate *Nodes* in the network. See also *Capacity*, *Load*.

Throughput The ratio of *Data Rate* being offered to a network by a *Source* or sources to the data rate being received by the corresponding *Destinations*.

Traffic Flow A flow of data over a *Connection* between a *Source* and its *Destination*.

Transmission Rate The rate of data transmission in bits per unit time. See also *Bit Rate*, *Data Rate*.

Transport Layer The communications layer that provides reliable, transparent transfer of data between end points; it provides *End-to-End* error recovery and *Flow Control*.

Utilisation The use of a *Link* for data transmission, averaged over a time interval. The value is often normalised, so that the value of 1 indicates that the link was completely utilised over the time interval.

Variance The non-uniformity of a set of samples from a constant value. Mathematically, the square of the set's standard deviation.

Window See *Sliding Window*.

Window Size The number of outstanding *Packets* or *Octets* in a *Sliding Window*.

Appendix B

Verifying TRUMP with Spin

The TRUMP transport protocol was designed primarily as a rate-based protocol which would fit into the rate-based congestion control framework described in this thesis. Other functionality apart from a rate-based flow control are secondary to the thesis. However, in the author's opinion, it is still essential that TRUMP be proved to be a correct and logically consistent protocol.

This appendix describes the validation of the TRUMP transport protocol using the Spin protocol verifier [Holzmann 91] [Holzmann 97] and its associated verification language, Promela. The Spin verifier is introduced, followed the Promela language and the verification capabilities of Spin. The implementation of TRUMP in Promela is then described, and the verification of this implementation using Spin is finally considered.

B.1 Introduction to Spin

The abstract from the 'Basic Spin Manual', part of the Spin software distribution, gives this description of Spin:

> Spin is a tool for analyzing the logical consistency of concurrent systems, specifically of data communication protocols. The system is described in a modeling language called Promela. The language allows for the dynamic creation of concurrent processes. Communication via message channels can be defined to be synchronous (i.e., rendezvous), or asynchronous (i.e., buffered).
>
> Given a model system specified in Promela, Spin can either perform random simulations of the system's execution or it can generate a C program that performs an efficient online verification of the system's correctness properties. During simulation and verification Spin checks for the absence of deadlocks, unspecified receptions, and unexecutable code. The verifier can also be used to verify the correctness of system invariants, it can find non-progress execution cycles, and it can verify correctness properties expressed in next-time free linear temporal logic formulae.
>
> The verifier is setup to be efficient and to use a minimal amount of memory. An exhaustive verification performed by Spin can establish with mathematical certainty whether or not a given behavior is error-free. Very large verification problems, that can ordinarily not be solved within the constraints of a given computer system, can be attacked with a frugal "bit state storage" technique, also known as *supertrace*. With

this method the state space can be collapsed to a small number of bits per reachable system state, with minimal side-effects.

B.2 The Promela Language

In order to be able to read and understand the implementation of TRUMP in the Promela language, given later in this appendix, an brief introduction to the language must first be given. The following is abstracted from the 'Basic Spin Manual', with additional text relating the discussion to the TRUMP code.

Promela is a verification modeling language. It provides a vehicle for making abstractions of protocols (or distributed systems in general) that suppress details that are unrelated to process interaction. The intended use of Spin is to verify fractions of process behavior, that for one reason or another are considered suspect. The relevant behavior is modeled in Promela and verified. A complete verification is therefore typically performed in a series of steps, with the construction of increasingly detailed Promela models. Each model can be verified with Spin under different types of assumptions about the environment (e.g., message loss, message duplications etc). Once the correctness of a model has been established with Spin, that fact can be used in the construction and verification of all subsequent models.

Promela programs consist of processes, message channels, and variables. Processes are global objects. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior, channels and global variables define the environment in which the processes run.

B.2.1 Executability

In Promela, there is no difference between conditions and statements: even isolated boolean conditions can be used as statements. The execution of every statement is conditional on its executability. Statements are either executable or blocked. The executability is the basic means of synchronization. A process can wait for an event to happen by waiting for a statement to become executable. A condition can only be executed (passed) when it holds. If the condition does not hold, execution blocks until it does.

B.2.2 Data Types

The primitive data types in Promela are `bool`, `byte`, `short`, `int` and `mtype`. A `bool` is a single bit of information. A `byte` is an unsigned quantity that can store a value between 0 and 255. `shorts` and `ints` are signed quantities that differ only in the range of values they can hold (typically 16-bits and 32-bits in size, respectively).

An `mtype` variable can be assigned symbolic values that are declared in an `mtype={...}` statement. This is the preferred way of specifying message types since it abstracts from the specific values to be used, and it makes the names of the constants available to an implementation, which can improve error reporting.

Variables can be also declared as arrays, using a C style syntax. An array of size N has elements numbered 0 to N-1. Element references outside of this range cause run-time errors in

Spin. C style structures can be created with the `typedef` operator: the TRUMP implementation uses a structure for the messages passed between the Sender and Receiver process. Fields within structures are accessed using the usual C style syntax.

B.2.3 Processes

The state of a variable or of a message channel can only be changed or inspected by a *process*. The behavior of a process is defined in a `proctype` declaration. The declaration body (enclosed in curly braces) consists of a list of zero or more declarations of local variables and/or statements. Promela accepts two different statement separators: an arrow '`->`'and the semicolon ';'. The two statement separators are equivalent. The arrow is used as an informal way to indicate a causal relation between two statements.

A proctype definition only declares process behavior, it does not execute it. When simulated with Spin, the `init` process is always instantiated. Statements in `init` can be used to instantiate other processes. In the TRUMP implementation, the Sender, Receiver and two link processes are instantiated via `init`.

B.2.4 Atomic Sequences

Unless blocked on a condition which is not yet true, a process can execute its statements. During verification, Spin will simulate all possible interleaving of the statements from the running processes. By prefixing a sequence of statements enclosed in curly braces with the keyword `d_step`, the user can indicate that the sequence is to be executed as one indivisible unit, non-interleaved with any other processes. It causes a run-time error if any statement in the sequence can block. `d_step` sequences in the TRUMP implementation are used to atomicise the complex manipulation of local variables.

B.2.5 Message Channels

Message channels are used to model the transfer of data from one process to another. They are declared either locally or globally, for instance as follows:

```
chan hold = [4] of { trumphdr };
```

This declares a channel that can store up to 4 messages of type `trumphdr` (a structure). Channel names can be passed from one process to another via parameters in process instantiations.

The statement `hold!expr` sends the value of variable `expr` to the channel just created: that is, it appends the value to the tail of the channel. `hold?msg` receives the message: it retrieves it from the head of the channel, and stores it in a variable `msg`. The channels pass messages in first-in-first-out order.

The send operation is executable only when the channel addressed is not full. The receive operation, similarly, is only executable when the channel is non empty. Nothing bad will happen if a statement happens to be non-executable. The process trying to execute it will be delayed until the statement, or, more likely, an alternative statement, becomes executable.

Several additional functions on channels are predefined, which allow a process to inspect the size of a channel's queue without blocking: `len(q)`, `empty(q)`, `nempty(q)`, `full(q)`

and `nfull(q)`, where `q` is the name of the channel. The latter four functions return true if the channel is empty, not empty, full, or not full, respectively.

B.2.6 Control Flow

Between the lines, we have already introduced three ways of defining control flow: concatenation of statements within a process, parallel execution of processes, and atomic sequences. There are three other control flow constructs in Promela to be discussed. They are case selection, repetition, and unconditional jumps.

The simplest construct is the selection structure. Using the relative values of two variables a and b to choose between two options, for instance, we can write:

```
if
:: (a != b) -> option1;
:: (a == b) -> option2;
fi;
```

The selection structure contains two execution sequences, each preceded by a double colon. Only one sequence from the list will be executed. A sequence can be selected only if its first statement is executable. The first statement is therefore called a *guard*.

In the above example the guards are mutually exclusive, but they need not be. If more than one guard is executable, one of the corresponding sequences is selected nondeterministically. If all guards are unexecutable the process will block until at least one of them can be selected. There is no restriction on the type of statements that can be used as a guard.

A logical extension of the selection structure is the repetition structure. As above, only one option can be selected for execution at a time. After the option completes, the execution of the structure is repeated.

```
do
:: (a != b) -> option1;
:: (a == b) -> option2;
od;
```

The normal way to terminate the repetition structure is with a `break` statement placed in one of the options. Note that to ensure the repetition is terminated, the guards on all the other options must be false, and hence not executable. Another way to break the repetition is with an unconditional jump: the infamous `goto` statement. The `goto label` statement is always executable, and takes execution to the named label elsewhere in the process.

One Promela statement commonly used with selection and repetition is the `skip` statement which is always executable and performs no action. A typical selection statement with no 'else' clause can be written as

```
if
:: (a == b) -> option2;
:: (a != b) -> skip;
fi;
```

Without the `skip` option, the selection would block the process until the guard `(a == b)` was indeed true. Promela also provides an `else` guard, which is only true if all other guard statements are false.

B.2.7 Promela Verification Support

An important language construct in Promela that needs little explanation is the `assert` statement. Statements of the form `assert(any_boolean_condition)` are always executable. If the boolean condition specified holds, the statement has no effect. If, however, the condition does not necessarily hold, the statement will produce an error report during verifications with Spin.

When Promela is used as a verification language the user must be able to make very specific assertions about the behavior that is being modeled. In particular, if a Promela is checked for the presence of deadlocks, the verifier must be able to distinguish a normal end state from an abnormal one.

A normal end state could be a state in which every Promela process that was instantiated has properly reached the end of the defining program body, and all message channels are empty. But, not all Promela process are, of course, meant to reach the end of their program body. Some may very well linger in an idle state, or they may sit patiently in a loop ready to spring into action when new input arrives.

To make it clear to the verifier that these alternate end states are legal, and do not constitute a deadlock, a Promela model can use end state *labels*. There may be more than one end state label per verification model. If so, all labels that occur within the same process body must be unique. The rule is that every label name that starts with the three character sequence "end" is an endstate label.

In the same spirit as the end state labels, the user can also define progress state labels. In this case, a progress state label will mark a state that must be executed for the protocol to make progress. Any infinite cycle in the protocol execution that does not pass through at least one of these progress states is a potential starvation loop. Again, all progress labels that occur within the same process body must be unique. The rule is that every label name that starts with the character sequence "progress" is an progress state label.

B.3 Spin Verification Capabilities

The verification capabilities of Spin are described in detail in [Holzmann 91] and in the 'Basic Spin Manual'. Again, the following brief description is abstracted from the latter.

Given a model system specified in Promela, Spin can either perform random simulations of the system's execution or it can generate a C program that performs a fast exhaustive verification of the system state space. The verifier can check, for instance, if user specified system invariants may be violated during a protocol's execution.

If invoked as

```
$ spin -a protocol.spin
```

Spin generates a protocol analyzer for the protocol described in the Promela language in the `protocol.spin` file. The output analyser is written into a set of C files that can be compiled to produce the analyzer (which is then executed to perform the analysis). To guarantee an exhaustive exploration of the state space, the program is compiled simply as

```
$ cc -o run pan.c
```

The executable program run can now be executed to perform the verification. The verification is truly exhaustive: it tests all possible event sequences in all possible orders. Exhaustive search works up to system state spaces of roughly 100,000 states. For larger systems this may, however, exhaust the available memory on the machine used.

The coverage of a conventional analysis goes down rapidly when the memory limit is hit, i.e. if there are twice as many states in the full state space than we can store, the effective coverage of the search is only 50% and so on. Spin does substantially better in those cases by using the bit state space storage method [Holzmann 90]. Large to very large systems can still be analyzed by using a memory efficient bit state space method by compiling the analyser as

```
$ cc -o run -DBITSTATE pan.c
```

An indication of the coverage of such a search can be derived from the hash factor given in the analyser's output. The hash factor is roughly equal to the maximum number of states divided by the actual number of states. A large hash factor (larger than 100) means, with high reliability, a coverage of 99% or 100%. As the hash factor approaches 1, the coverage approaches 0%.

Note carefully that the analyzer realizes a partial coverage only in cases where traditional verifiers are either unable to perform a search, or realize a far smaller coverage. In no case will Spin produce an answer that is less reliable than that produced by other automated verification systems (quite on the contrary).

B.4 Implementing TRUMP in Promela

TRUMP is implemented in Promela as several intercommunicating processes. The Sender process attempts to deliver data packets successfully to the Receiver, following the protocol rules for connection setup, data transmission and connection teardown. The Receiver process receives data packets from the Sender, and returns acknowledgment packets. The Sender and Receiver are connected via two unidirectional links, whose operations are performed by two instantiations of the Badlinksrc process.

As the goal here is to verify the corrctness of the TRUMP protocol, features of TRUMP which do not affect its protocol operation are not implemented. This includes the protocol's rate-based flow control, the security features of TRUMP, and the use of connection identifiers. These features have no impact on the logical correctness and consistency of the protocol.

Protocol features of TRUMP which have not been implemented are the qualities of service and packet fragmentation/reassembly. The reason for these omissions is the limit on the time available to complete this thesis. However, despite this, the majority of the TRUMP protocol has been implemented in Promela and verified by the Spin program.

Although TRUMP is not as large as a 'heavyweight' protocol such as TCP, it is still a complex protocol, and cannot be exhaustively verified using the platforms available to the author, mainly due to memory size considerations. To prove that TRUMP is indeed correct where only some of the features are available, the implementation of TRUMP in Promela is controlled by macro processor[1] 'define's which enable or disable certain features:

REORD: Allow packet reordering on the two links betwen the Sender and the Receiver. The maximum number of reordering operations per link is given by the value MAXREORD in a `constants` file. The maximum number of packets 'buffered' by the link for reordering is given by the value MAXPKTS in the `constants` file.

LOSE: Allow packet loss on the two links betwen the Sender and the Receiver. The maximum number of packet losses per link is given by the value MAXLOSSES in the `constants` file.

SELACK: Allow selective acknowledgments to be used between the Sender and the Receiver. The number of packets acknowledged in each selective acknowledgment is given by the value SELACKSIZE in the `constants` file.

RECVERRS: Allow the Receiver to send errors in Special Acknowledgment at any time.

IMPLICIT: Allow implicit connection setups and teardowns as well as the default explicit connection setups and teardowns.

ASSERT: Install many Promela `assert()` statements into the TRUMP implementation.

The **ASSERT** option adds assertions to TRUMP and does not increase its complexity. The **RECVERRS** and **IMPLICIT** options add many extra states to both the Sender and Receiver processes, and increase the size of the state space to be exhaustively searched and verified. The same is true for the **SELACK** option, but this also increases the size of the inter-process messages and the number of local variables, which again increases the state space. The **REORD** and **LOSE** options do not affect the Sender or Receiver, but the packet reorderings and losses greatly add to the size of the state space to be searched.

B.4.1 TRUMP's Promela Implementation

TRUMP's implementation in Promela is divided into two files: the actual implementation is contained in `trump.spin`, and the constants used by TRUMP are contained in the `constants` file. This file is short and is given below.

```
/*****************************/
/** Easily-changed constants **/
/*****************************/
#define DATAPKTS        1       /* Send this many data packets */

#ifdef REORD
# define MAXPKTS        4       /* Max packets buffered by reordering */
# define MAXREORD       20      /* Max number of reorders permitted */
#endif

#ifdef LOSE
```

[1] The macro processor used is the C preprocessor.

```
# define MAXLOSSES      20      /* Maximum number of losses permitted */
#endif
```

```
#ifdef SELACK
# define SELACKSIZE     8       /* Number of acks in selack packet */
# define EMPTY          255     /* Ack pkt is empty if seq_base==EMPTY */
#endif

/* The Sender keeps a linked list of
 * the in-transit packets. Each element
 *  has one of the following values.
 */
#define NOTSENT         253     /* Packet has not been transmitted yet */
#define RECEIVED        254     /* Packet was sucessfully received */
#define NULL            255     /* Null pointer */
```

The implementation of TRUMP in `trump.spin` is so heavily affected by macro processor directives as to be virtually unreadable. The version of the file below gives the values of the macro processor options, the code after macro processing, and with line numbers.

Some notes should be first given on the TRUMP implementation. In order to determine which packet to transmit next (and hence its sequence number), the TRUMP Sender needs to keep a linked list of in-transit packets and not-yet-transmitted packets. Promela does not have the capability of representing arbitrarily-sized linked lists. To overcome this deficiency, the Sender declares an array of bytes, `data_status`, to store the list of in-transit packets.

Two byte variables, `head` and `tail`, point at the head and tail nodes of the list. An arbitrary value of 255 is chosen to represent the NULL pointer. Elements in the array hold the index number of the next element in the array. The tail node has the value NULL.

There will obviously be some array elements which are not in transit to the Receiver. These have either the value NOTSENT (253) if they have not been transmitted yet, or the value RECEIVED (254) if they were sucessfully received by the Receiver.

As values in the `data_status` array above 252 have special meanings, there cannot be elements in the array with index 252 or above. Therefore, the TRUMP implementation in Promela is limited to around 250 data packets.

Here, finally, is the implementation of TRUMP in `trump.spin`.

```
 1  /**
 2   ** TRUMP verification in Promela
 3   **
 4   ** Copyright (c) 1997 Warren Toomey
 5   ** Revision: 1.32
 6   ** Date: 1997/04/17
 7   **/
 8
 9
10  /************************/
11  /** Source-code defines **/
12  /************************/
13
14  #define REORD                           /* Allow packet reordering on links */
15  #define LOSE                            /* Allow packet loss on links */
16  #define SELACK                          /* Allow selective acknowledgments */
```

```
17  #define RECVERRS                        /* Allow receiver to send errors */
18  #define IMPLICIT                        /* Allow implicit setups/teardowns */
19  #define ASSERT                          /* Install lots of assertions */
20
21  #include "constants"                    /* Get constants used below */
22
23
24  /*******************/
25  /** TRUMP Headers **/
26  /*******************/
27
28  /* Each packet must be one of these types */
29  mtype = { setup, sack, data, ack, teardown, datasyn }
30
31
32  /* A TRUMP packet has the following headers */
33  typedef trumphdr {
34      mtype pkt_type;                     /* Type of packet this is */
35      byte  seq_no;                       /* Sequence number of data segment */
36                                          /* in selack[0] */
37      byte  seq_top;                      /* Top seq# in selack bitmap */
38      bool  selack[SELACKSIZE];           /* Selack bitmap */
39      bool  explicit_down;                /* If true, do explicit teardowns */
40      bool  error_val;                    /* Error value of reply */
41  }
42
43  /************************/
44  /* Channel Definitions */
45  /************************/
46
47  chan sendtolink = [1] of { trumphdr }   /* Channel from src to badlink  */
48  chan recvtolink = [1] of { trumphdr }   /* Channel from dest to badlink */
49  chan linktorecv = [1] of { trumphdr }   /* Channel from badlink to dest */
50  chan linktosend = [1] of { trumphdr }   /* Channel from badlink to src  */
51
52
53  /*********************/
54  /** Network Section **/
55  /*********************/
56
57  proctype Badnetwork(chan in, out)
58  {
59    trumphdr packet;                      /* Packet received on link */
60
61    chan hold = [MAXPKTS] of { trumphdr }; /* Pkt buffer for reordering */
62    xr hold; xs hold;
63    byte reordcnt;                        /* Count of # of reorderings done */
64
65
66    byte losecnt;                         /* Count of # of losses done */
67    losecnt=0;                            /* Initialise loss count */
68    reordcnt=0;                           /* Initialise reordering count */
69
70  end_bad0: /* Forerver, do */
71          do
72          :: in?packet ->         /* Receive a packet from in channel */
```

```
 73                 if
 74                 :: out!packet   /* Retransmit it immediately, */
 75
 76                 :: ((reordcnt<MAXREORD) && nfull(hold)) ->
 77                     hold!packet; reordcnt++; /* save it for later transmit, */
 78
 79                 :: (losecnt<MAXLOSSES) -> losecnt++; /* or lose it entirely */
 80                 fi;
 81
 82         :: hold?packet ->
 83                 out!packet      /* Reinsert old data if no link pkt */
 84         od;
 85  }
 86
 87
 88  /******************/
 89  /** TRUMP Sender **/
 90  /******************/
 91
 92  /* The sender has the following main states:
 93   *
 94   *  transmitting setup packets (optional)
 95   *  transmitting data packets
 96   *  transmitting teardown packets
 97   *  finished!
 98   */
 99
100  proctype Sender(chan send_in, send_out)
101  {
102   trumphdr a;                    /* Packet received from destination */
103   byte data_status[DATAPKTS];    /* List of in-transit seq nums */
104   byte head;                     /* Head of list */
105   byte tail;                     /* Tail of list */
106   byte prev;                     /* Previous node pointer */
107   byte lost;                     /* Highest known lost packet */
108   byte notsent;                  /* Lowest not-yet sent packet */
109   byte i;                        /* Loop counter */
110   byte S;                        /* Next sequence number to send */
111   bool explicit_conn;            /* Should we do an explicit connection? */
112   bool explicit_down;            /* Should we do an explicit teardown? */
113
114
115   /* Initialise the data_status array to all NOTSENT */
116   d_step {
117     i=0; S=0; head=NULL; tail=NULL; lost=NULL; notsent=0;
118     do
119     :: (i==DATAPKTS) -> break;
120     :: (i!=DATAPKTS) -> data_status[i]= NOTSENT; i++;
121     od;
122   }
123
124   /* Choose to do an implicit or explicit connection */
125   if
126   :: (1) -> explicit_conn=0;                    /* Implicit connection */
127             if
128             :: (1) -> explicit_down=0;          /* And implicit teardown, or */
```

```
129             :: (1) -> explicit_down=1;           /* Explicit teardown */
130             fi;
131
132             /* Send off the implicit conn packet, which is also 1st data */
133             d_step {
134              a.pkt_type=datasyn; a.error_val=0; a.seq_no=0;
135              a.explicit_down=explicit_down; head=0; tail=0; lost=NULL;
136              data_status[0]=NULL; notsent=1;
137             };
138
139             send_out!a; goto progress_send0;     /* Move to state 2, data tx */
140   :: (1) -> explicit_conn=1; explicit_down=1;    /* Explicit connection */
141   fi;
142
143
144   /* State 1: sending setup packets */
145   do
146   :: send_in?a -> /* Receive a packet from the receiver */
147           if /* it's a sack with no error, move to state 2 */
148           :: (a.pkt_type==sack && a.error_val==0) -> break;
149
150              /* it's a sack with an error, terminate now */
151           :: (a.pkt_type==sack && a.error_val!=0) -> goto end_exit_state;
152
153              /* some other packet, ignore it! */
154           :: else ->
155                 assert(0);      /* Should not get any other type of packet */
156           fi;
157
158          /* If we can't receive a packet from the receiver */
159          /* then send it a setup packet */
160   :: (empty(send_in) && nfull(send_out)) ->
161            d_step { a.pkt_type=setup; a.seq_no=0; a.error_val=0; }
162            send_out!a;
163   od;
164
165  progress_send0:
166
167
168   /* State 2: sending data packets */
169   do
170   :: send_in?a; /* Receive a packet from the receiver */
171           if /* it's a sack with no error, ignore it */
172           :: (a.pkt_type==sack && a.error_val==0) -> skip;
173
174              /* it's a sack with an error, terminate now */
175           :: (a.pkt_type==sack && a.error_val!=0) -> goto end_exit_state;
176
177              /* it's an ack packet, update the data_status array */
178           :: (a.pkt_type == ack) ->
179                 d_step {
180                   printf("Sender got SELACK %d/%d \n",a.seq_no,a.seq_top);
181                   i=a.seq_no; S=0; printf("\t");
182                   do
183                   :: (i>a.seq_top) -> break;
184                   :: else -> printf("%d ", a.selack[S]); i++; S++;
```

```
185                   od; printf("\n");
186                 }
187
188                 /* Assert that  seq_no <= seq_top < seq_no+SELACKSIZE */
189                 /* We could assert that no ack bits above seq_top are on, */
190                 /* but the sender ignores them anyway */
191                 assert(a.seq_no<=a.seq_top);
192                 assert(a.seq_top<(a.seq_no+SELACKSIZE));
193
194                 /* If it's an ack for the datasyn, check the explicit_down */
195                 d_step {
196                   if
197                   :: ((explicit_conn==0)&&(a.seq_no==0)) ->
198                     if
199                     :: ((explicit_down==0)&&(a.explicit_down==0)) -> skip;
200                     :: ((explicit_down==0)&&(a.explicit_down==1)) ->
201                         explicit_down=1; /* Recvr asked for explicit teardown */
202
203                     :: ((explicit_down==1)&&(a.explicit_down==0)) ->
204
205                         /* Receiver asked for implicit teardown, but we've */
206                         /* already chosen explicit teardown, error! */
207                         assert(0);
208                     :: ((explicit_down==1)&&(a.explicit_down==1)) -> skip;
209                     fi;
210                   :: else -> skip;
211                   fi;
212                 }
213
214                 /* Deal with each sequence number in turn */
215                 d_step {
216                    S=0;         /* S points at 0th bit in bitmap */
217                    /* For the selack code, loop a.seq_no up to a.seq_top */
218                    do
219                    :: (a.seq_no > a.seq_top) -> break;
220                    :: else ->
221
222                         /* Assert that we did in fact transmit this one */
223                         assert(data_status[a.seq_no]!=NOTSENT);
224
225                         if
226                         :: (a.selack[S]==0) ->  skip; /* Ignore lost pkts */
227                         :: (a.selack[S]!=0) ->  /* it was received! */
228
229                         /* Assert that we did in fact transmit this one */
230                                 assert(data_status[a.seq_no]!=NOTSENT);
231
232                                 if
233                                 :: (head==NULL) -> skip; /* Already got it */
234                                 :: (data_status[a.seq_no]==RECEIVED) -> skip;
235
236                                 :: (head==a.seq_no) -> /* it's head packet */
237                                     /* Move the head up */
238                                     head= data_status[a.seq_no];
239
240                                     if /* Null lost ptr if we received it */
```

```
241                                     :: (lost==a.seq_no) -> lost=NULL;
242                                     :: else -> skip;
243                                     fi;
244
245                                     if /* list now empty? */
246                                     :: (head==NULL) -> tail=NULL;
247                                     :: else -> skip;
248                                     fi;
249                                 :: else ->
250                                     /* Not the head! Find previous node */
251                                     prev=head;
252
253                                     do  /* prev can't point at NULL! */
254                                     :: (data_status[prev]==NULL) -> assert(0);
255                                     :: (data_status[prev]==a.seq_no) -> break;
256                                     :: else -> prev= data_status[prev];
257                                     od;
258
259                                     if /* Null lost ptr if we received it */
260                                     :: (lost==a.seq_no) -> lost=prev;
261                                     :: else -> skip;
262                                     fi;
263
264                                     /* Point lost at prev if prev's */
265                                     /* seq_no < a.seq_no */
266                                     if
267                                     :: (prev<a.seq_no) -> lost=prev;
268                                     :: else -> skip;
269                                     fi;
270                                     /* Now, remove seq_no from list */
271                                     data_status[prev]=data_status[a.seq_no];
272                                 fi;
273                                 if /* seq_no was the tail, point tail at prev */
274                                 :: (a.seq_no==tail) -> tail=prev;
275                                 :: else -> skip;
276                                 fi;
277
278                                 /* Mark the sequence number as RECEIVED */
279                                 data_status[a.seq_no]=RECEIVED;
280                         fi;
281                         a.seq_no++; S++;        /* Move up to the next seq_no */
282                     od;
283                 };
284             /* some other packet, error! */
285          :: else -> assert(0);
286          fi;
287
288         /* If we can't receive a packet from the receiver */
289         /* then send it a data packet */
290   :: (empty(send_in) && nfull(send_out)) ->
291
292           /* Complicated bit. To obtain the next sequence number: */
293           /*   If lost!=NULL, choose head of intransit list, else */
294           /*   Use lowest notsent packet */
295           d_step {
296             S= NULL;            /* Assume no packets left to send */
```

```
297
298             if /* Lost can't point to anything is list is empty! */
299             :: ((head==NULL) && (lost!=NULL)) -> assert(0);
300             :: else -> skip;
301             fi;
302
303             if  /* no packets left to send */
304             :: ((lost==NULL)&&(head==NULL)&&(notsent==DATAPKTS)) -> skip;
305
306             :: ((lost==NULL)&&(notsent<DATAPKTS)) -> /* choose new seq# */
307                 S= notsent;             /* Notsent has lowest notsent seq# */
308
309                 if                      /* Append notsent to tail */
310                 :: (tail==NULL) -> head=notsent;
311                 :: (tail!=NULL) -> data_status[tail]=notsent;
312                 fi;
313                 tail=notsent; data_status[tail]=NULL; notsent++; /* Link ops */
314
315             :: else ->          /* Something must be left in intransit list */
316
317                 S= head;
318
319                 if              /* Remove head node */
320                 :: (head==tail) -> head=NULL; tail=NULL;
321
322                 :: (head!=tail) ->
323                         assert(head!=NULL); assert(tail!=NULL);
324                         head= data_status[head];
325                 fi;
326
327                 if                      /* Append S to tail */
328                 :: (tail==NULL) -> head=S; tail=S;
329
330                 :: (tail!=NULL) -> data_status[tail]=S; tail=S;
331                 fi;
332
333                 data_status[tail]=NULL;
334
335                 if /* we're retxing the highest lost packet */
336                 :: (lost==S) -> lost=NULL;      /* No lost pkts left now */
337                 :: else -> skip;
338                 fi;
339             fi;
340           } /* End of the d_step many lines ago! */
341
342           /* If S is still NULL here, no pkts to tx, then move to next state */
343           if
344           :: (S==NULL) -> break;
345           :: else -> skip;
346           fi;
347
348           d_step { /* otherwise build and transmit the data packet */
349             a.pkt_type=data; a.error_val=0; a.seq_no=S;
350           }
351           send_out!a;
352   od;
```

```
353
354  progress_send1:
355    /* Assert that all packets marked as RECEIVED */
356    i=0;
357    do
358    :: (i==DATAPKTS) -> break;
359    :: (i!=DATAPKTS) -> assert(data_status[i]==RECEIVED); i++;
360    od;
361
362    if /* we don't need to do an explicit teardown, skip the teardown state */
363    :: (explicit_down==0) -> goto end_exit_state2;
364    :: else -> skip;      /* we do, so go straight into it */
365    fi;
366
367
368   /* State N: sending teardown packets */
369   do
370   :: send_in?a -> /* we can receive a packet from the receiver */
371           if /* it's a teardown, move to exit state */
372           :: (a.pkt_type==teardown) -> break;
373
374              /* some other packet, ignore it! */
375           :: (a.pkt_type != teardown) -> skip;
376           fi;
377
378        /* If we can't receive a packet from the receiver */
379        /* then send it a teardown packet */
380   :: (empty(send_in) && nfull(send_out)) ->
381           d_step { a.pkt_type=teardown; a.seq_no=0; a.error_val=0; }
382
383           send_out!a;
384   od;
385
386  end_exit_state:
387   printf("Sender exiting\n");
388
389  /* Sit here mopping up any packets */
390  end_exit_state2:
391   do
392   :: send_in?a;
393   od;
394  }
395
396
397  /********************/
398  /** TRUMP Receiver **/
399  /********************/
400
401  proctype Receiver(chan recv_in, recv_out)
402  {
403   trumphdr b;                    /* Received packet */
404   byte i;                        /* Loop counter */
405   trumphdr a;                    /* Acknowledgment packet */
406   bool alreadygot;               /* True if already marked in selack packet */
407   bool explicit_down;            /* Expect an explicit teardown */
408   bool gotit[DATAPKTS];          /* True if we've got this seq# */
```

```
409   byte got_thismany;            /* Count of seq#s we've got (excl. dups) */
410
411                                 /* Initialise the empty acknowledgment packet */
412   d_step {
413      a.seq_no= EMPTY;
414      a.pkt_type=ack;
415
416      i=0;                       /* Initialise the gotit array */
417      got_thismany=0;
418      do
419      :: (i==DATAPKTS) -> break;
420      :: (i!=DATAPKTS) -> gotit[i]=0; i++;
421      od;
422   };
423
424                 /* State 1: receiving setup packets */
425   do
426   :: recv_in?b;
427      if
428      :: (b.pkt_type==setup) -> /* If pkt is a setup pkt, return sack pkt */
429          d_step { b.pkt_type=sack; b.seq_no=0; b.error_val=0; explicit_down=1; }
430
431          recv_out!b; break;
432
433      :: (b.pkt_type==setup) -> /* If pkt is a setup pkt, return ERROR sack pkt */
434          d_step { b.pkt_type=sack; b.seq_no=0; b.error_val=1; }
435
436          recv_out!b;
437          goto end_recv_tosspkts; /* and just sink remaining packets */
438
439      :: (b.pkt_type==datasyn) -> /* Implicit conn pkt */
440          explicit_down=b.explicit_down; /* Keep explicit teardown flag */
441
442          /* Choose for explicit or implicit teardown */
443          if
444          :: (b.explicit_down==1) -> skip;
445          :: (b.explicit_down==0) -> explicit_down=0;     /* Implicit */
446          :: (b.explicit_down==0) -> explicit_down=1;     /* Explicit */
447          fi;
448          goto recv_gotdata; /* Skip to the data receive state */
449
450      :: else -> skip;    /* If not a setup packet, ignore it */
451      fi;
452    od;
453
454   progress_recv0:
455                 /* State 2: receiving data packets */
456    do
457
458    :: (got_thismany==DATAPKTS) ->
459
460   end_recvimplicit:
461      recv_in?b; goto process_data;       /* End state if got all seq#s */
462    :: (got_thismany!=DATAPKTS) ->         /* otherwise it isn't */
463
464      recv_in?b;
```

```
465
466  process_data:
467      if
468      :: (b.pkt_type==setup) -> /* If pkt is a setup pkt, return sack pkt */
469          d_step { b.pkt_type=sack; b.seq_no=0; b.error_val=0; }
470          recv_out!b;
471
472      :: (b.pkt_type==setup) -> /* If pkt is a setup pkt, return ERROR pkt */
473          d_step { b.pkt_type=sack; b.seq_no=0; b.error_val=1; }
474          recv_out!b;
475          goto end_recv_tosspkts; /* go and sink remaining packets */
476
477      :: (b.pkt_type==data) -> /* Received a data packet */
478  recv_gotdata:
479                                  /* Mark seq# as seen */
480          if
481          :: (gotit[b.seq_no]==1) -> skip;
482                                  /* & count the # unique seq#s we've received */
483          :: (gotit[b.seq_no]==0) -> gotit[b.seq_no]=1; got_thismany++;
484          fi;
485
486          /* First thing is to see if we can fit this seq_no */
487          /* into the existing acknowledgment packet */
488          if
489          :: ((a.seq_no!=EMPTY) && ((b.seq_no<a.seq_no)
490              || (b.seq_no>=a.seq_no+SELACKSIZE))) ->
491                  /* Seq_no doesn't fit, so transmit existing ack packet */
492                  recv_out!a;
493
494                  /* Reinitialise the ack packet */
495                  d_step { a.seq_no= EMPTY; }
496          :: else -> skip;
497          fi;
498
499          /* At this point, either the ack packet is EMPTY or we can */
500          /* fit the seq_no in */
501          d_step {
502            if
503            :: (a.seq_no==EMPTY) ->
504                  a.seq_no= b.seq_no; a.seq_top= b.seq_no;
505                  a.selack[0]=1; alreadygot=0;
506                  a.explicit_down=explicit_down;
507
508            :: (a.seq_no!=EMPTY) ->
509                  i = b.seq_no - a.seq_no;
510                  alreadygot=a.selack[i]; a.selack[i]=1;
511                  if
512                  :: (b.seq_no>a.seq_top) -> a.seq_top=b.seq_no;
513                  :: else -> skip;
514                  fi;
515            fi;
516          }
517
518          /* If data pkt duplicates one we already have, send ack pkt back NOW */
519          if
520          :: (alreadygot==1) ->
```

```
521                 recv_out!a; a.seq_no=EMPTY;
522         :: else -> skip;
523         fi;
524
525     :: (b.pkt_type==teardown) -> /* Received a teardown packet */
526
527         /* It's an error if implicit teardowns selected */
528         assert(explicit_down==1);
529
530         /* Reflect the teardown packet */
531         d_step { b.pkt_type=teardown; b.seq_no=0; b.error_val=0; }
532         recv_out!b; break;
533
534     :: (b.pkt_type==data) -> /* If pkt is a data pkt, return ERROR sack pkt */
535         d_step { b.pkt_type=sack; b.error_val=1; } /* (to be perverse) */
536
537         recv_out!b; goto end_recv_tosspkts;
538
539     :: else -> skip;    /* Not a setup, data or teardown packet, ignore it */
540     fi;
541   od;
542
543
544                 /* State 3: Got a teardown, mopping up */
545 end_recv0:
546  do
547  :: recv_in?b;
548     if
549     :: (b.pkt_type==teardown) -> /* Reflect teardowns to source */
550         d_step { b.pkt_type=teardown; b.seq_no=0; b.error_val=0; }
551         recv_out!b;
552
553     :: else -> skip;    /* If not a setup or teardown packet, ignore it */
554     fi;
555  od;
556
557                 /* State for collecting remaining packets */
558 end_recv_tosspkts:
559   do
560   :: recv_in?b; d_step { b.pkt_type=sack; b.error_val=1; } recv_out!b;
561   od;
562 }
563
564
565 /***************************/
566 /** System Initialisation **/
567 /***************************/
568
569 init
570 {
571   printf("TRUMP verification in Promela\n");
572   run Sender(linktosend, sendtolink);
573   run Badnetwork(sendtolink, linktorecv);
574   run Badnetwork(recvtolink, linktosend);
575   run Receiver(linktorecv, recvtolink);
576 }
```

B.5 TRUMP State Diagrams

Much of the Promela code for TRUMP deals with the manipulation of internal variables to detect packet loss and duplication, and to ensure packet retransmission when required. This tends to obscure the operation of TRUMP as a protocol, i.e as a communications language between two peers. To expose the operation of the protocol itself, state diagrams for the TRUMP sender and receiver are presented: these emphasise the communication operations in TRUMP. States and transitions dealing with the 'internal' functionality of TRUMP (e.g working on the linked list of in-transit packets) have been omitted to improve clarity.

The two diagrams show the states and state transitions for the TRUMP sender and receiver. States are represented as nodes with line numbers corresponding to the implementation of TRUMP given in the previous section: essentially, these correspond to the points where a choice is available, or where a implementation can be blocked on a receive or send operation. Receiving states are marked in blue, and sending states are marked in pink. Transitions are represented as directed edges, with labels describing the reason for the transition. Transitions marked with a '`*`' character must be taken.

B.5.1 TRUMP Sender State Diagram

The TRUMP Sender operation goes through three stages. Connection establishment negotiates connection parameters and determines if the receiver is willing to accept connection. Data transmission either transmits a data packet if no acknowledgment packet can be received, or deals with the ack packet. Connection termination occurs on receiver errors, or if all data packets have been received sucessfully.

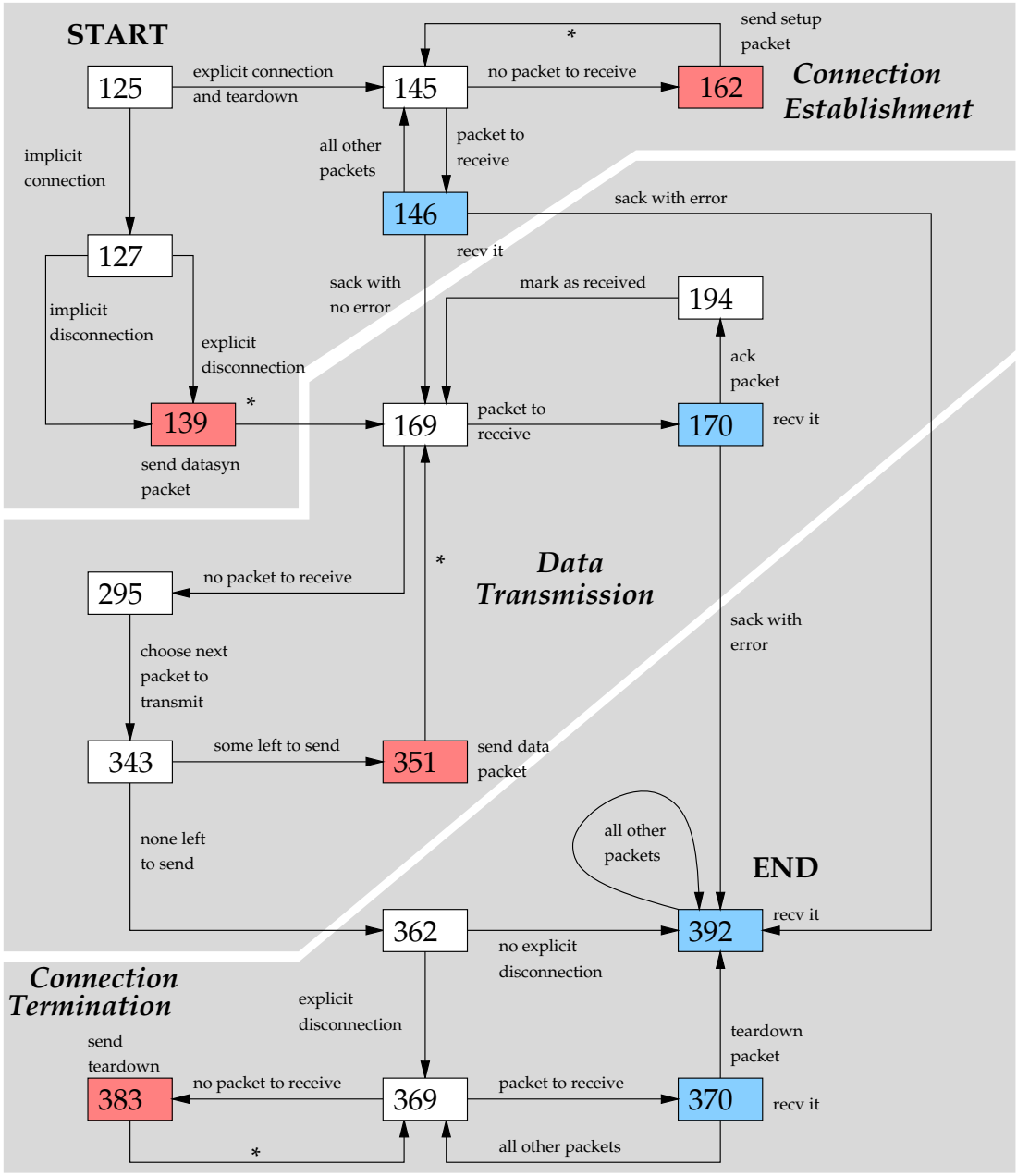


Figure 64: TRUMP Sender State Diagram

B.5.2 TRUMP Receiver State Diagram

The TRUMP Receiver operation also goes through three stages. Connection establishment negotiates connection parameters and determines if the receiver is willing to accept connection. Data reception is controlled by packets from the source (lines 456 and 464), and returns acknowledgment packets. Connection termination occurs when teardown packets are received, and returns teardown packets to the source. A fourth stage of operation, Abnormal Termination, is only used if the receiver wishes to abort the connection at any stage.

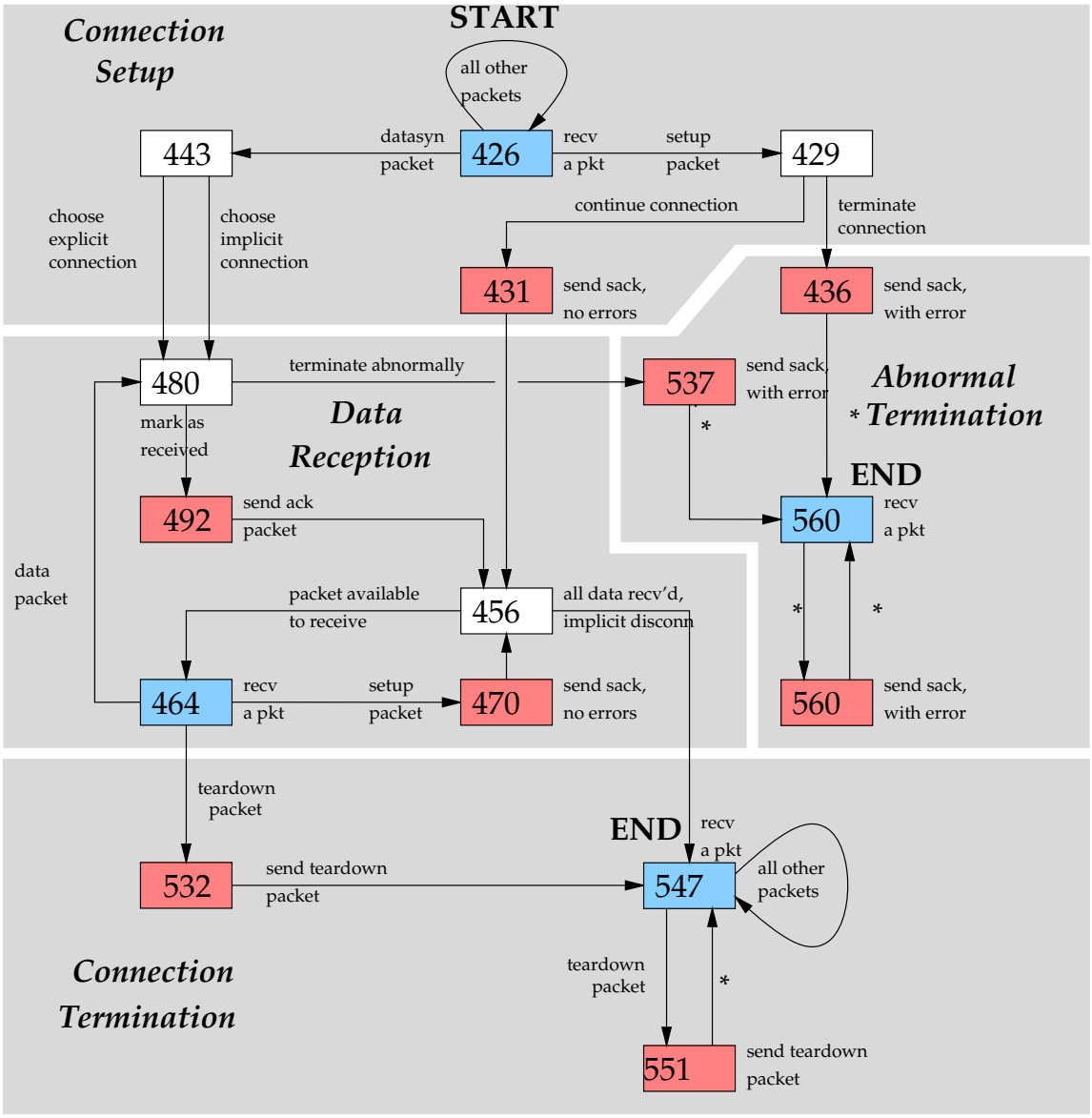


Figure 65: TRUMP Receiver State Diagram

B.6 Verification of TRUMP

A TRUMP protocol, with certain of the available features enabled, is correct if:

- both the Sender and Receiver finish in an 'end' state,
- there are no messages left to be received in any message channel,
- each process has passed through all of its 'progress' states, and
- all assertions have been shown to be true,

for **all** possible interleavings of statements and events across all the processes in the implementation: this forms the *state space* to be verfied.

The state space must also incorporate the state of all local variables within each process, as state transitions may be chosen on the basis of these state variables. In the implementation of TRUMP in Promela, both the sender and receiver have several local variables, of which the largest are the arrays used to track the number of packets sucessfully transmitted.

In the verification of the TRUMP protocol with Spin, the number of possible states, combined with the size of each state, makes an exhaustive proof of the protocol impossible. The computing equipment available does not have enough real memory to hold the state space. Using the 'supertrace' method of verification, the amount of memory required to hold each state is much reduced, but again the number of states is too large to fully prove the protocol. However, let us look at the areas in which the protocol can be proved.

B.6.1 No Features Enabled

In Section B.4, it was mentioned that certain TRUMP features could be enabled or disabled via preprocessor macros. Each feature adds extra complexity and state size to TRUMP, and for some of the combinations of features, the TRUMP protocol can be verified.

With no features enabled, the protocol provides reliable data transmission of N data packets across an error-free link with no selective acknowledgments (i.e one acknowledgment per data packet) using explicit connection setup and teardown. For values of N up to 28, the protocol can be exhaustively proved correct: there are 177,614 states (of size 128 bytes) and 222,136 state transitions for N =28.

Above N =28, we must use the 'supertrace' method of verification in Spin. Here, the verifier can estimate the fraction of all states verified by consulting the number of unused hash entries and the number of hash collisions for a hash table that is used to represent the state space. Using the supertrace method, a protocol can be proved correct for the fraction of possible states covered by the verifier. In the case of Spin, a verification shows that a protocol is correct after covering either '$\geq$ 99.9% of states', '$\geq$ 98% of states' or '$<$ 98% of states'.

Using supertrace, TRUMP with no features enabled is provably correct after covering $\geq$ 98% of states for N up to 250 data packets.

B.6.2 Selective Acknowledgments

With the enabling of selective acknowledgments of size 8 (that is, one acknowledgment for 8 data packets), the state vector is 60 bytes larger then for the equivalent 'no features' state, and the number of states is approximately 4 times that of the equivalent 'no features' protocol. The supertrace method was used for all values of N, showing that TRUMP with selective acknowlegments of size 8 is provably correct after covering $\geq$ 98% of states for N up to 250 data packets.

B.6.3 Receiver Errors

One optional feature of the TRUMP implementation in Promela is to allow the receiver to return a special 'error' acknowledgment at any time during the course of the connection setup, data transmission and connection teardown. Upon receipt of this acknowledgment, the sender must immediately desist transmission of all packets. This arbitrary return of an 'error' acknowledgment increases the size of the state space to approximately 2.6 times that of the equivalent 'no

features' protocol; the state size is unchanged. The supertrace method was used for all values of N, showing that TRUMP with arbitrary receiver errors is provably correct after covering $\geq 98\%$ of states for N up to 250 data packets.

B.6.4 Implicit Connection Setup

Enabling implicit connection setups and teardown (with the latter selectable by both sender and receiver), the size of each state is approximately twice that with implicit connections disabled. The state space for implicit connections is also approximately 3.5 times that of the equivalent 'no features' protocol. The supertrace method was used for all values of N, showing that TRUMP with implicit connections is provably correct after covering $\geq 98\%$ of states for N up to 250 data packets.

B.6.5 Packet Losses

Two of the selectable 'features' in the TRUMP implementation are the enabling of packet reordering on the two links between the sender and receiver, and the enabling of packet loss on these two links. These features do not affect the sender's or receiver's code, but they do affect the protocol's operation. Both options have a tremendous effect on the possible interleavings of the sender, receiver and link states, and so significantly affect the size of the total state space to be verified. With either enabled, TRUMP cannot be verified for large values of N.

The implementation of TRUMP in Promela imposes a limit on the number of packet reorderings and packet losses on a single link. This is set in the 'constants' file at 20 reorderings and 20 losses per link. Even with these small values, the state space is extremely large.

With only packet loss enabled, the state space to be explored is *at least* 330 times that of TRUMP with no packet loss, for equivalent values of N: for example, there are $2.8 * 10^7$ states for N =5 packets. The size of each state is unaffected. Figure 66 shows the range of N vs. packet loss values for which the TRUMP specification in Promela can be verified correct.

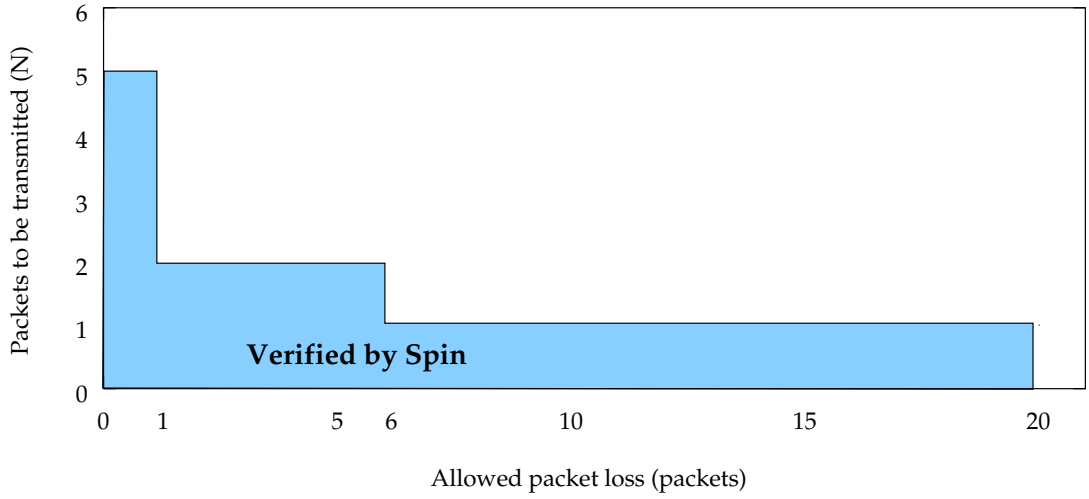


Figure 66: Verification of TRUMP during Packet Loss

B.6.6 Packet Reordering

With only packet reorderings enabled, the state space to be explored is again very much larger than that of TRUMP with no packet loss. The size of each state is unaffected. Figure 67 shows the range of N vs. packet loss values for which the TRUMP specification in Promela can be verified correct.

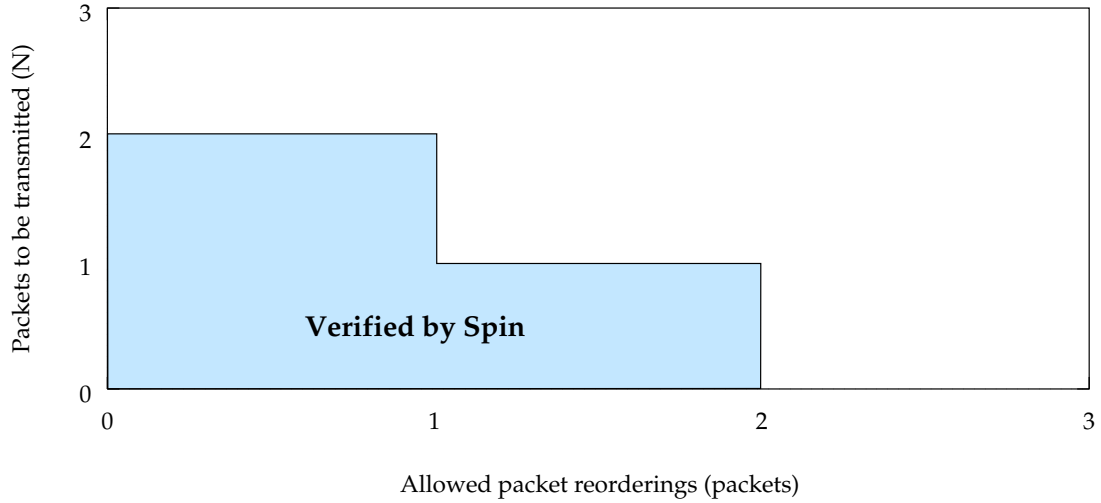


Figure 67: Verification of TRUMP during Packet Reordering

B.7 Summary

The correct operation of a complex transport protocol must be verified; otherwise, severe operational errors may come to light either after an implementation of the protocol, or after the implementation's deployment. In this appendix, we have reviewed the use of the Spin protocol verification tool to prove TRUMP correct where possible.

Due to CPU and memory constraints, Spin was unable to exhaustively prove TRUMP correct. The Promela implementation of the protocol was designed to allow certain TRUMP functionality to be enabled and disabled. With packet loss and packet reordering disabled, the TRUMP protocol was verified with the following features enabled, for up to 250 data packets per connection:

- No other features,
- Selective acknowledgments of size 8,
- Arbitrary receiver errors, and
- Implicit connection setup.

The complexity of the Promela implementation of TRUMP increases dramatically with packet losses or packet reordering enabled. With no features enabled, Spin was able to prove the correctness of the TRUMP protocol for a very small number of packet loss/data packet combinations, and similarly for a very small number of packet reordering/data packet combinations.

Although TRUMP has not been exhaustively verified, it appears to be a robust transport protocol, with no correctness flaws discovered.

Appendix C

TRUMP Protocol Headers

TRUMP is a transport protocol which provides for the reliable delivery of data across an unreliable packet-switched network. Chapter 5 gives a feature-oriented examination of the TRUMP transport protocol. This appendix describes a proposed set of TRUMP headers for use in an IPv6 network.

In the following section, all fields are unsigned integers or, if single bits, boolean flags. All multi-octet fields in TRUMP's headers are stored in big-endian format. All diagrams are 4 octets wide. For fields where not all the possible values are enumerated, the undefined values are currently invalid and reserved for future use. Fields that are described as 'unused' should be set to all bits off.

The first 64 bits of each TRUMP header are nearly identical for all packet types, and are shown in the following figure. A description of each field follows.

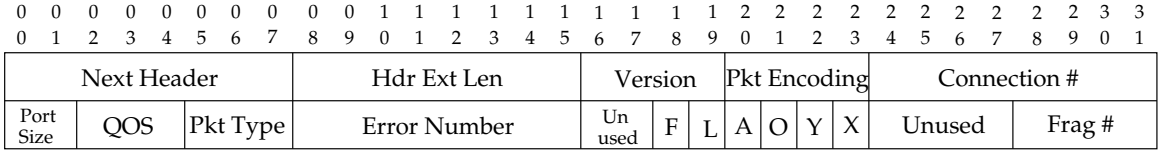


Figure 68: First 32 bits in TRUMP Headers

Next Hdr Part of the IPv6 header specification, described in [Stallings 96].

Hdr Extension Length Part of the IPv6 header specification, described in [Stallings 96].

Version The version of the transport protocol. The current TRUMP version is 2. Future versions of TRUMP will have higher version numbers.

Packet Encoding The encoding of the packet from the **Connection Number** field onwards. This is predominately used to allow for encryption of transport protocol packets. Currently, only two values out of the sixteen possible are defined:

0 The header and data are in plain-text.

1 The header (with the exception of the first four fields) and data are encrypted in DES block mode, using a key negotiated by the source and destination endpoints. The method of key negotiation is not a part of the TRUMP protocol. The checksum is not encrypted, and is set to the checksum of the whole packet after encryption.

Connection Number Currently, this field is not used and reserved. In future versions of TRUMP, it will hold a number that uniquely identifies the (source, destination, source port, destination port) tuple.

Port Size Identifies the size of the endpoint identifiers. The four possible values are:

0 Endpoint identifiers are 0 bits in size.
1 Endpoint identifiers are 16 bits in size.
2 Endpoint identifiers are 32 bits in size.
3 Endpoint identifiers are 64 bits in size.

The latter two are designed for use in distributed systems, where large integers can be used to store such things as object descriptors and capabilities.

Quality of Service Used in **SETUP** packets and in **DATA** packets where the **Y** bit is set. The field is invalid at any other time, and must be set to all bits off. The field holds the requested quality of service for the connection. Two values are currently defined:

0 Reliable. The destination sends ACKs and the source retransmits.
1 Time-sensitive. The destination sends ACKs but the source does not retransmit lost data.

Packet Type The type of packet. Five values are currently defined:

0 A SETUP packet.
1 A DATA packet.
2 An ACK packet.
3 A SACK packet.
4 A TEARDOWN packet.

Error Indication of any error that has occurred. This field is only valid for a SACK packet, and must be set to all bits off in any other packet. Defined values are given below.

Bits 50 to 55 are used to hold bit fields for particular packet types. The fields shown are described below. Refer to the rest of this section for details of the other fields in each packet type.

A – Return ACK Immediately Used in **DATA** packets. If set, the destination must return an ACK packet for this DATA packet immediately. This field is illegal in other packets.

O – Last segment Used in **DATA** packets. If set, the segment in the DATA packet is the last in the message. This field is illegal in other packets.

Y – DATA is Implicit SETUP Used in **DATA** packets. If set, the DATA packet holds the first segment of the message, and is an implicit SETUP. The QOS field is not valid in a DATA packet if this is not set. This field is illegal in other packets.

X – Explicit Teardown Used in **SETUP** and **SACK** packets. If set, the source must perform an explicit connection teardown. This field is illegal in other packets.

In addition, there are fields in each **DATA** packet that are used if a source must fragment a segment.

F – Segment is a Fragment If set, the segment is a fragment of another segment. The segment sequence number is the same as the original segment. This field is illegal in non-DATA packets.

L – Last Fragment If set, the segment is the last fragment of another segment. This field is illegal in non-DATA packets.

Fragment Number The fragment number of the given fragment. If **F** is not set, this field is invalid. This field is illegal in non-DATA packets.

C.1 Connection Setup

Explicit connection setup in TRUMP is performed by transmission of a **SETUP** packet from source to destination, acknowledged by a **SACK** from destination to source. This exchange passes the message identification, source and destination port, and type of connection tear-down to the destination, and returns flow control information or a connection failure error to the source.

The **SETUP** packet has the fields as shown in the following figure.

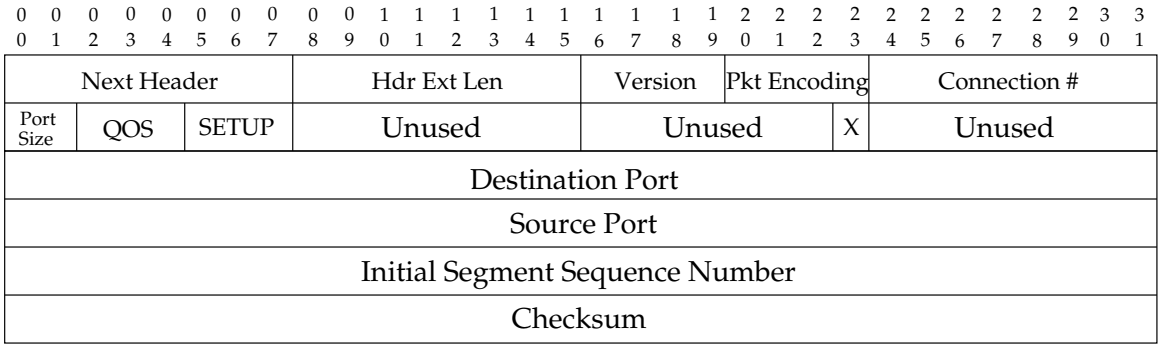


Figure 69: Connection Setup Packet

Destination Port The TRUMP destination port of the connection.

Source Port The TRUMP source port of the connection.

Initial Segment Sequence Number The first segment in the message will have this sequence number.

Checksum A 32-bit checksum calculated over the fields in the packet. This is calculated using the algorithm described in [Fletcher 82].

In all TRUMP packets, the **Source Port** and **Destination Ports** are variable-sized, from zero-to 64-bits. The figures shown above and following give packets where these fields are 32-bits in size.

C.2 Special Acknowledgment

Explicit connection setup in TRUMP is acknowledged by a **SACK** packet. This packet is also used to abnormally terminate a message transmission (i.e an established connection).

The **SACK** packet has the fields as shown in the following figure.

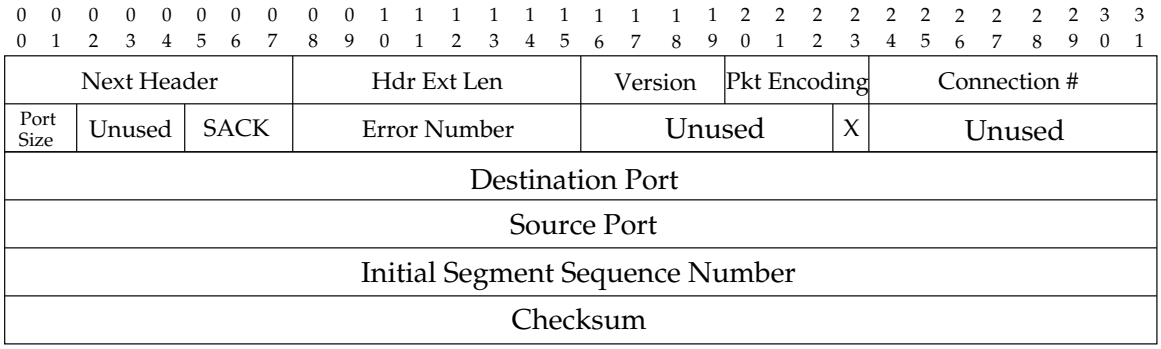


Figure 70: Special Acknowledgment Packet

The **SACK** packet has the following extra fields:

Error Indication of any error that has occurred. Defined values are:

00 No error has occurred.

01 The destination port doesn't exist.

02 The destination port is currently unavailable.

03 The source port doesn't have enough privileges for the connection.

04 The source machine doesn't have enough privileges for the connection.

05 The destination port has been destroyed.

06 The destination cannot continue the connection.

07 The destination TRUMP implementation cannot cope with the requested port size.

08 The destination TRUMP implementation cannot cope with the requested encryption scheme.

09 The destination TRUMP implementation cannot cope with the requested quality of service.

10 The last packet from the source had a port size different than that specified in the initial **SETUP** packet.

Destination Port The TRUMP destination port of the connection.

Source Port The TRUMP source port of the connection.

Checksum A 32-bit checksum calculated over the fields in the packet. This is calculated using the algorithm described in [Fletcher 82].

C.3 Data Transmission

The **DATA** packet is used to transport data from the source to the destination.

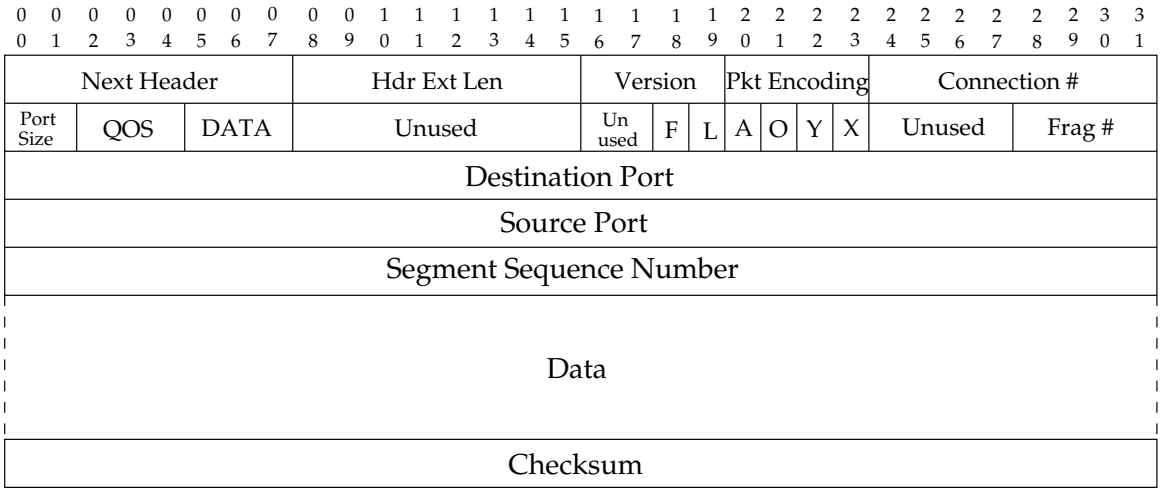


Figure 71: Data Transmission Packet

The **DATA** packet has the following extra fields:

Destination Port The TRUMP destination port of the connection.

Source Port The TRUMP source port of the connection.

Segment Sequence Number The sequence number of the segment in the DATA field.

Data The actual payload of the packet, a TRUMP message segment.

Checksum A 32-bit checksum calculated over the fields in the packet. This is calculated using the algorithm described in [Fletcher 82].

C.4 Data Acknowledgment

The **ACK** packet is used to acknowledge data from the source.

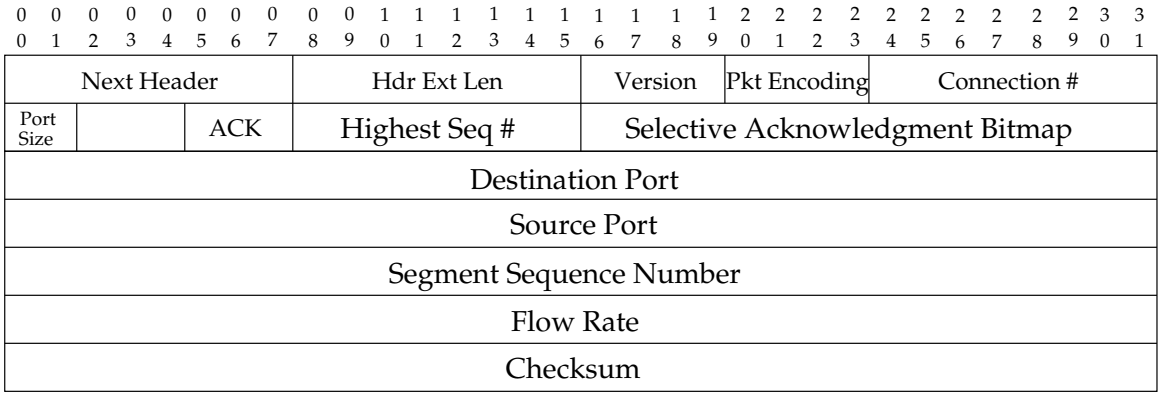


Figure 72: Data Acknowledgment Packet

The **ACK** packet has the following extra fields:

Destination Port The TRUMP destination port of the connection.

Source Port The TRUMP source port of the connection.

Segment Sequence Number The sequence number of the first segment being acknowledged by this packet.

ACK Bitmap The bitmap that selectively acknowledges up to 16 segments. Assume the Segment Sequence Number is N. If segment S was received correctly, bit 2^{S-N} is set on; otherwise, it is set to zero.

Highest Sequence Number The highest sequence number H positively or negatively received in the bitmap, relative to the Segment Sequence Number N. Sequence numbers $> N + H$ are not acknowledged in the bitmap.

Flow Rate The highest bit rate of segment transmission by the source which will not cause buffer overflows in the destination.

Checksum A 32-bit checksum calculated over the fields in the packet. This is calculated using the algorithm described in [Fletcher 82].

C.5 Connection Termination

The **TEARDOWN** packet is used to explicitly close a finished message (i.e a completed connection). This allows the destination to remove data structures that it might have kept in order to detect delayed, out of sequence, or duplicate packets.

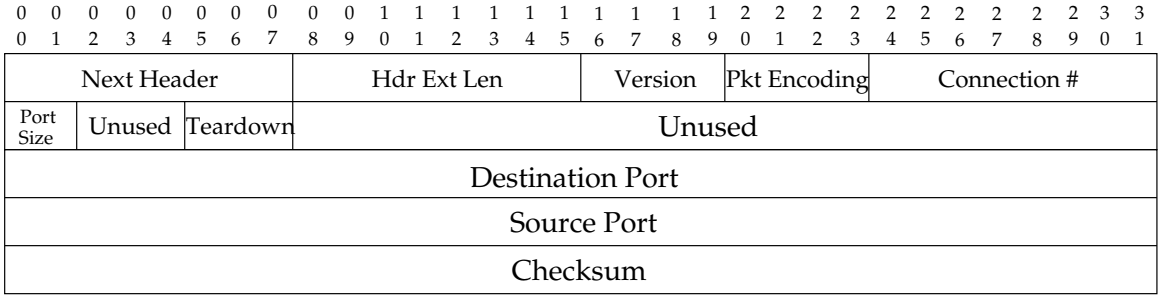


Figure 73: Connection Close Packet

The **TEARDOWN** header has the following fields:

Destination Port The TRUMP destination port of the connection.

Source Port The TRUMP source port of the connection.

Checksum A 32-bit checksum calculated over the fields in the packet. This is calculated using the algorithm described in [Fletcher 82].

Appendix D

Complexity of RBCC

D.1 Introduction

As the rate-based congestion control framework proposed in this thesis shifts the majority of control into the routers within the network, the cost of this extra workload needs to be examined. In order for the framework to be practical, the benefits of the new control framework must outweigh the drawbacks of implementing it.

This appendix details the operations of RBCC within a router. To begin with, the data structures for RBCC are explored, indicating the memory requirements for the implementation of RBCC in a router. Following this, each specific operation of RBCC, its complexity and cost, will be described.

D.2 Estimation of Flows through a Router

The overhead of RBCC on a router, in terms of both memory and CPU requirements, depends greatly upon the number of individual traffic flows passing through it at any given time.

[Apisdorf *et al* 96] describe a hardware-based OC3 network monitor for the NSF's very high speed Backbone Network Service. One of the functions of the network monitor is to measure the number of flows on an OC3 link. The definition of a traffic flow from the paper is given below.

> We specifically do not restrict ourselves to the TCP connection definition, i.e., SYN/FIN-based, of a flow. Instead, we define a flow based on traffic satisfying specified temporal and spatial locality conditions, as observed at an internal point of the network. ... That is, a flow represents actual traffic activity from one or both of its transmission endpoints as perceived at a given network measurement point. A flow is active as long as observed packets that meet the flow specification arrive separated in time by less than a specified timeout value...
>
> We explored a range of timeouts ..., and found that 64 seconds was a reasonable compromise between the size of the flow table and the amount of work setting up and tearing down flows between the same points...
>
> We define flows as unidirectional, i.e., bidirectional traffic between A and B would show up as two separate flows: traffic from A to B, and traffic from B to A.

In [Apisdorf *et al* 96], one graph shows the number of flows "on an OC3 trunk of MCI's IP backbone during the period between 29 July and 1 August 1996", and is reproduced in Figure 74.

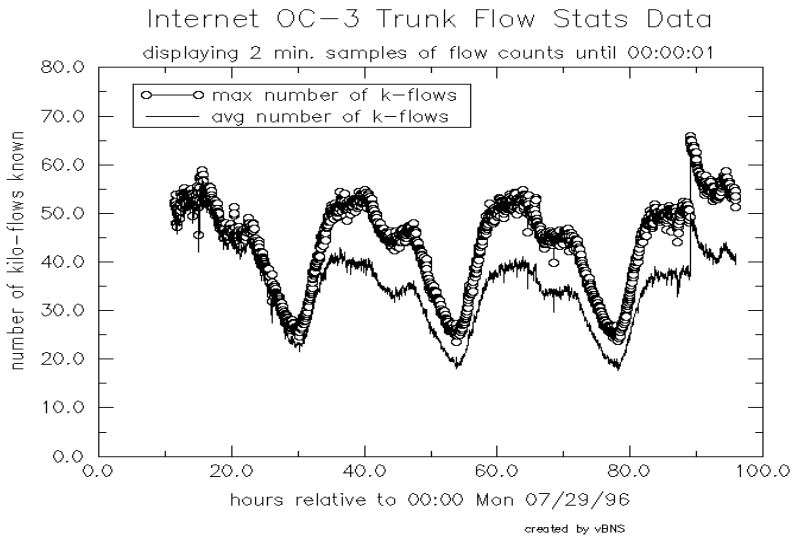


Figure 74: Flows over 2-minute intervals on MCI IP OC3 backbone trunk

The graph shows that, during the trace period, the router sustained a peak of 67,000 traffic flows over a period of 2 minutes. Although this is a single data point, it indicates the order of magnitude for the number of flows across a core Internet router in late 1996.

D.3 Size of RBCC Data Structures

The details of each traffic flow held by RBCC within a router are organised as per-interface 'Allocation Bandwidth Tables'. The term 'table' is a misnomer, as a linear list or array would most likely not be used. Instead, an implementation would choose a fast-search structure like a hash table or a B-tree. This would minimise search time of the table on receipt of a packet.

An individual flow's node within the table would include the following fields:

```
struct conv_type                /* Table entry for a single flow */
{
                                /* Congestion fields */
    double   flow_id;           /* Unique identifier for this flow */
    int      bottlenode;        /* Node which is the bottleneck */
    double   rate;              /* Rate allocated by a node */
    double   desired_rate;      /* Rate desired by the source */
    double   lim_rate;          /* Limiting rate on the flow */

                                  /* Cached packet fields */
    double   cache_rate;          /* Rate measured along the route */
    double   cache_desired_rate; /* Rate desired by the source */
    int      cache_bottlenode;   /* node that last set the rate field */

                                /* Data structure fields */
    struct conv_type *next;     /* Pointer to next flow in the list */
```

```
    struct conv_type *left;      /* Pointer to left node in B-tree */
    struct conv_type *right;     /* Pointer to right node in B-tree */
};
```

The node belongs to two data structures:

- a singly-linked list ordered by the value of the `lim_rate` field, as described in Section 6.2.3, and linked by the `next` field; and
- a B-tree ordered by the value of the `flow_id` field, and linked by the `left` and `right` fields.

On a machine where integers are 32-bits, pointers are 32-bits, and doubles are 32-bits, as described in Section 7.3, the node would occupy 44 bytes. On the router described in the previous section, at least 3 megabytes of memory would be required to hold all of the per-interface ABT tables.

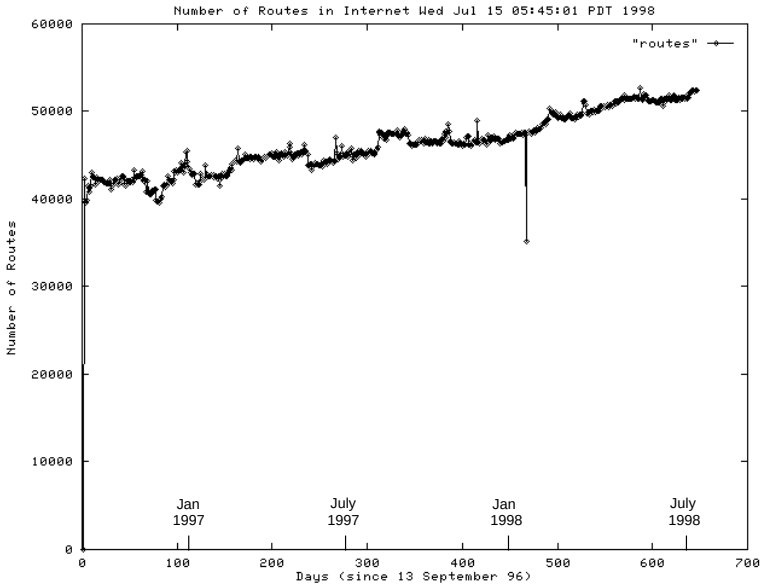


Figure 75: Routes in the Internet from September 1996

This compares well with the storage space required for the routing table in current Internet routers. The number of routes currently in the Internet is shown in Figure 75, taken from [Bates 98]. Current Internet routers typically store their routing table in a modified Patricia tree [Morrison 68]. Using a straightforward implementation of Patricia trees for a 32-bit processor, with 24 bytes for leaves and 24-bytes for internal nodes [Degermark *et al* 97], a routing table of 52,000 entries would occupy 2.5 megabytes of memory.

On current core Internet routers with 52,000 routing entries and 67,000 simultaneous traffic flows, the routing table and RBCC ABT storage costs are commensurate.

D.4 Packet Arrival for an Existing Flow

The CPU overhead of RBCC will be examined in this and following sections. Again, this overhead is related to the number of flows in the router's ABT tables. We will first consider the case

of a packet arriving at a router which has an existing node with the Allocated Bandwidth Table; in other words, it is part of an already-recognised traffic flow. Packets for new flows will be addresses in the next section. Let us examine the RBCC operations that the router will perform.

RBCC is used after the router has has determined to which output interface the packet must be sent for retransmission, and before it is queued for transmission. The set of operations which RBCC must perform are:

1. Find data structures for the flow to which packet relates

The router must search the per-interface ABT B-tree for the given `flow_id` in the packet. The order of complexity of this operation will be $O(\log_2 N)$, with N the number of flows in the B-tree (*not* the overall number of flows through the router). This is the same complexity required to find the correct route, and hence the correct output interface, for the packet [Morrison 68].

The result of this operation is a pointer `c`, which points to the flow's entry in the per-interface ABT table.

2. Determine if packet's congestion fields are already cached.

This can be achieved with a bitwise comparison of the three congestion fields `rate`, `desired_rate` and `bottlenode`, between packet and cached values. The cost of this operation is O(1) and insignificant.

If the packet's congestion fields are found in the cache, then we may proceed to step 5.

3. Forward update decisions.

This is described in detail in Section 6.2.2. Typically, 0.6% of packets reach this point, as is shown in Section 11.10.

```
if (pkt->desired_rate != c->cache_desired_rate) {
        c->cache_desired_rate = pkt->desired_rate;
        c->cache_bottlenode = pkt->bottlenode;
        c->cache_rate = pkt->rate;
        must perform ABT update;
}

if (we are not pkt->bottlenode and (pkt->rate != c->cache_rate)) {
        c->cache_desired_rate = pkt->desired_rate;
        c->cache_bottlenode = pkt->bottlenode;
        c->cache_rate = pkt->rate;
        must perform ABT update;
}

if (pkt->rate < c->cache_rate)
        c->cache_desired_rate = pkt->desired_rate;
        c->cache_bottlenode = pkt->bottlenode;
        must perform ABT update;
}
```

Again, this operation is O(1) and insignificant. Common code inside the three test above can be aggregated to further reduce the CPU overhead. If we do not need to perform ABT updates, then we may proceed to step 5. As described in Section 11.4, reverse updates appear to be detrimental, and are therefore not considered here.

4. Perform ABT update.

This is described in detail in Section 6.2.3. Typically, 0.3% of packets reach this point, as is shown in Section 11.10. The algorithm given in Section 6.2.3 is reproduced here with the comments removed.

```
bandwidth_left= bandwidth; flows_left= flowcount;
for (c= head_of_flow_list; c!=NULL; c= c->next) {
    allocation = bandwidth_left / (1.0 * flows_left);

    if (c->lim_rate > allocation) {
        c->rate= allocation; c->bottlenode=node;
    } else {
        allocation= c->lim_rate; c->rate= allocation;
    }

    bandwidth_left-= allocation; flows_left--;
}
```

This operation is O(N), with N the number of flows in the per-interface B-tree (*not* the overall number of flows through the router). However, the amount of work inside the loop is small.

5. Update congestion control fields.

This is described in detail in Section 6.2.4. All packets must pass through this operation.

```
if ((pkt->rate > c->rate) || (we are the bottlenode)) {
  pkt->rate = c->rate; pkt->bottlenode = c->bottlenode;
}
```

Again, this operation is O(1) and insignificant.

Operations 2 and 5 are O(1), and are done to all incoming packets. Operation 3 is O(1) and is performed on 0.6% of all incoming packets. Operation 1, a B-tree search, is $O(\log_2 N)$, but each search step is short. Operation 4, the core of the RBCC algorithm, is O(N), is the operation with the highest cost.

If N, the number of nodes in each per-interface ABT, is smaller than the number of routes known by the router, then the cost of the RBCC algorithm (i.e operations 1 to 5) will be commensurate with, if not less than, the cost of determining the packet's route and output interface.

D.5 Packet Arrival for a New Flow

On receipt of a packet for a traffic flow which is not yet represented in any ABT within the router, a new flow node must be constructed and inserted in the appropriate table.

The construction pseudo-code is:

```
create a new node pointed to by conv;
conv->flow_id= pkt->flow_id;
conv->desired_rate = pkt->desired_rate;
conv->rate = pkt->rate;

/* Work out the correct bottle node */
if (pkt->rate == infinity) { conv->bottlenode = us; }
else { conv->bottlenode = pkt->bottlenode; }

/* Work out the limiting rate */
conv->lim_rate = pkt->desired_rate;
if ((pkt->bottlenode != us) && (pkt->rate < pkt->desired_rate))
    conv->lim_rate = pkt->rate;

/* Cache the packet's fields */
c->cache_desired_rate = pkt->desired_rate;
c->cache_bottlenode = pkt->bottlenode;
c->cache_rate = pkt->rate;
```

The router must then perform two list insertions. The node must be inserted into the B-tree on the basis of its `flow_id` value: this is an O($\log_2$N) operation. The node must also be inserted into the singly-linked list, ordered by the `lim_rate` value: this is an O(N) operation.

D.6 Removal of an Existing Flow

A flow node must be removed from the appropriate ABT when a traffic flow terminates normally (as described in Section 6.3.4), or when the loss of a flow is detected (as described in Section 6.3.5).

The removal of the node from the B-tree is an O($\log_2$N) operation. The removal of the node from the singly-linked list is an an O(N) operation; however, if a doubly-linked list was maintained instead, this operation would become an O(1) operation.

D.7 Route Changes

Current routing tables within Internet routers tend not to change faster than once per second [Stanford 96]. If the output interface for a routing entry changes, then a router using RBCC must find the affected ABT entries, remove them from one per-interface ABT data structure, and insert them into a second per-interface ABT data structure. Although this is an O(N) operation, its frequency of occurrence is much lower then the operations considered previously.

D.8 Summary

The rate-based congestion control framework proposed in this thesis, and RBCC in particular, does place a higher load on routers within the network. Extra memory is required to hold the ABT data structures for RBCC, and for current core Internet routers, this is on the order of several megabytes of memory.

Of greater concern is the added processing burden that RBCC places on the router. Most operations are O(1) and negligible, or $O(\log_2 N)$ and similar to the cost of finding the output interface for a received packet. Operation 4 in Section D.4 has the highest order of complexity, O(N), and thus the highest cost. This cost is mitigated by the small number of packets which undergo this operation (0.3%), the small amount of code in Operation 4, and that N is the number of traffic flows exiting a single output interface on the router.

A full implementation of RBCC in a real network router is required to accurately determine the full cost of the proposed rate-based congestion control framework. The estimations in this appendix, however, indicate that costs of RBCC are comparable to the current packet forwarding operations in Internet routers.

Appendix E

REAL NetLanguage Files for Simulated Scenarios

Here are the most relevant parts of the REAL input files for the network scenarios described in Chapter 9. I have removed some mandatory lines that control the underlying NEST simulator on which REAL is based. In each file given, the keywords SRC, DST and ROUTER must be replaced with appropriate function names. The triplets required are given below:

TRUMP/RBCC: trump, trsink, rbcc_router, respectively.

TCP Tahoe: jk_tahoe, sink, router, respectively.

TCP Reno: jk_reno, sink, router, respectively.

TCP Vegas: vegas, sink, router, respectively.

All scenarios use the following global parameters:

```
real_params {
        scale_factor = 1.0;                     # Scale time to seconds
        ack_size = 40;                          # ACK size in octets
        random_seed = 23;                       # Seed to random()
        buffer_size = 15000;                    # Output iface buf size (bytes)
        ftp_pkt_size = 1500;                    # Data packet size
        ftp_window = 600;                       # Max window size in pkts
        selack_size= 1;                         # Size of selective ack bitmap
        decongestion_mechanism = 1;             # Drop Tail
        sch_policy = 1;                         # First Come, First Served
        util_time= 2.0;                         # Measure link util every 2sec
}
```

Also note that the 'thresh 5 0.75' parameter to the router is ignored by the existing REAL router code, and is only used by the RBCC router code.

E.1 Scenario 1

```
nodes{
        default{
                function = SRC;                 # Source
                start_time = 0,1000;            # Start at 1 millisecond
                plot = true;                    # Plot this node
                num_pkts = 1000;                # Num packets to send
                }

        node 1 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 2 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 3 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 4 { function = DST; }
        node 5 { dest = 4; }
}

# Bandwidth in bps, latency in microseconds
edges{
        default { bandwidth = 10000000; latency = 1; }
        { 1 -> 2; }
        { 2 -> 3; bandwidth = 1000000; latency = 30; }
        { 4 -> 3; }
        { 5 -> 1; }
}
```

E.2 Scenario 2

```
nodes{
        default{
                function = SRC;                 # Source
                start_time = 0,100;             # Start at 0 + 100 microseconds
                plot = true;                    # Plot this node
                num_pkts = 1000;                # Num packets to send
                }

        node 1 { function = ROUTER; params= 'thresh 5 0.75'; }
        node 2 { function = ROUTER; params= 'thresh 5 0.75'; }
        node 3 { function = ROUTER; params= 'thresh 5 0.75'; }
        node 4 { function = ROUTER; params= 'thresh 5 0.75'; }
        node 9 { function = DST; }
        node 10 { function = DST; }
        node 11 { function = DST; }
        node 12 { function = DST; }

        node 5 { dest = 10; }
        node 6 { start_time = 140,0; dest = 11; num_pkts= 100; }
        node 7 { start_time = 12,0; dest = 12;}
        node 8 { start_time = 18,0; dest = 9;}
}

# Bandwidth in bps, latency in microseconds
edges{
        default{ bandwidth = 10000000; latency = 1; }
```

```
        { 1 -> 2; }
        { 2 -> 3; bandwidth = 64000; latency = 1000000;}
        { 3 -> 4; }
        { 5 -> 1; }
        { 6 -> 1; }
        { 7 -> 1; }
        { 8 -> 3; }
        { 9 -> 2; }
        { 10 -> 3; }
        { 11 -> 4; }
        { 12 -> 4; }
}
```

Scenario 2a is the same as Scenario 2, but sources 6 and 7 have finite desired bandwidths of 18kbps and 32kbps, respectively; these are set in the corresponding node lines with the options `peak = 18000.0;` and `peak = 32000.0;`.

E.3 Scenario 3

```
nodes{
        default{
                function = SRC;
                start_time = 0,100;             # Start at 0 + 100 microseconds
                plot = true;                    # Plot this node
                num_pkts = 1000;                # Num packets to send
                }

        node 1 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 2 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 3 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 4 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 5 { function = ROUTER; params= 'thresh 5 0.75';  }

        node 10 { function = DST; }
        node 11 { function = DST; }
        node 12 { function = DST; }
        node 13 { function = DST; }

        node 6 { dest = 11; }
        node 9 { start_time = 5,0;  dest = 10; }
        node 7 { start_time = 10,0; dest = 12; num_pkts= 100; }
        node 8 { start_time = 15,0; dest = 13; num_pkts= 100; }
}

# Bandwidth in bps, latency in microseconds
edges{
        default{ bandwidth = 10000000; latency = 1; }
        { 1 -> 2; }
        { 2 -> 3; bandwidth = 512000; latency = 1000000;}
        { 3 -> 4; bandwidth = 128000; latency = 1000000;}
        { 4 -> 5; bandwidth = 64000;  latency = 1000000;}

        { 6 -> 1; }
```

```
        { 7 -> 3; }
        { 8 -> 4; }
        { 9 -> 4; }

        { 10 -> 2; }
        { 11 -> 5; }
        { 12 -> 5; }
        { 13 -> 5; }
}
```

E.4 Scenario 4

```
nodes{
        default{
                function = SRC;
                plot = false;                   # Don't plot this node
                num_pkts = 1000;                # Num packets to send
                }

        node 1 { start_time= 1,0; dest = 9; plot=true; }
        node 2 { start_time= 2,0; dest = 10; plot=true; }
        node 3 { start_time= 3,0; dest = 11; plot=true; }
        node 4 { start_time= 4,0; dest = 12; plot=true; }
        node 5 { start_time= 5,0; dest = 13; plot=true; }
        node 6 { start_time= 6,0; dest = 14; plot=true; }
        node 7 { start_time= 7,0; dest = 15; plot=true; }
        node 8 { start_time= 8,0; dest = 16; plot=true; }
        node 9 { function = DST;  plot=true; }
        node 10 { function = DST; plot=true; }
        node 11 { function = DST; plot=true; }
        node 12 { function = DST; plot=true; }
        node 13 { function = DST; plot=true; }
        node 14 { function = DST; plot=true; }
        node 15 { function = DST; plot=true; }
        node 16 { function = DST; plot=true; }

        node 17 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 18 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 19 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 20 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 21 { function = ROUTER; plot=true; params= 'thresh 5 0.75';  }
        node 22 { function = ROUTER; plot=true; params= 'thresh 5 0.75';  }
        node 23 { function = ROUTER; plot=true; params= 'thresh 5 0.75';  }
        node 24 { function = ROUTER; plot=true; params= 'thresh 5 0.75';  }
        node 25 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 26 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 27 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 28 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 29 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 30 { function = ROUTER; params= 'thresh 5 0.75';  }
}

# Bandwidth in bps, latency in microseconds
edges{
```

```
        default{ bandwidth = 10000000; latency = 1; }
        { 1 -> 17; }
        { 2 -> 17; }
        { 3 -> 18; }
        { 4 -> 18; }
        { 5 -> 19; }
        { 6 -> 19; }
        { 7 -> 20; }
        { 8 -> 20; }
        { 17 -> 21; }
        { 18 -> 21; }
        { 19 -> 22; }
        { 20 -> 22; }
        { 21 -> 23; }
        { 22 -> 23; }
        { 23 -> 24; bandwidth= 64000; latency= 1000000; }
        { 24 -> 25; }
        { 24 -> 26; }
        { 25 -> 27; }
        { 25 -> 28; }
        { 26 -> 29; }
        { 26 -> 30; }
        { 9 -> 27; }
        { 10 -> 27; }
        { 13 -> 28; }
        { 14 -> 28; }
        { 11 -> 29; }
        { 12 -> 29; }
        { 15 -> 30; }
        { 16 -> 30; }
}
```

E.5 Scenario 5

```
nodes{
        default{
                function = SRC;
                plot = true;
                num_pkts = 1000;                # Num packets to send
                }

        node 6 { start_time= 0,0;   dest = 18; }
        node 7 { start_time= 20,0;  dest = 24; }
        node 8 { start_time= 50,0;  dest = 21; num_pkts = 100; }
        node 9 { start_time= 60,0;  dest = 25; }
        node 10 { start_time= 61,0; dest = 23; }
        node 11 { start_time= 1,0;  dest = 16; }
        node 12 { start_time= 21,0; dest = 19; num_pkts = 100; }
        node 13 { start_time= 40,0; dest = 20; }
        node 14 { start_time= 41,0; dest = 22; num_pkts = 100; }
        node 15 { start_time= 49,0; dest = 17; }

        node 16 { function = DST; }
        node 17 { function = DST; }
        node 18 { function = DST; }
```

```
        node 19 { function = DST; }
        node 20 { function = DST; }
        node 21 { function = DST; }
        node 22 { function = DST; }
        node 23 { function = DST; }
        node 24 { function = DST; }
        node 25 { function = DST; }

        node 1 { function = ROUTER; plot=false; params= 'thresh 5 0.75';  }
        node 2 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 3 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 4 { function = ROUTER; params= 'thresh 5 0.75';  }
        node 5 { function = ROUTER; plot=false; params= 'thresh 5 0.75';  }
}

# Bandwidth in bps, latency in microseconds
edges{
        default { bandwidth = 10000000; latency = 10; }
        { 6 -> 1; }
        { 7 -> 1; }
        { 16 -> 1; }
        { 17 -> 1; }
        { 8 -> 2; }
        { 9 -> 2; }
        { 18 -> 2; }
        { 19 -> 2; }
        { 10 -> 3; }
        { 11 -> 3; }
        { 20 -> 3; }
        { 21 -> 3; }
        { 12 -> 4; }
        { 13 -> 4; }
        { 22 -> 4; }
        { 23 -> 4; }
        { 14 -> 5; }
        { 15 -> 5; }
        { 24 -> 5; }
        { 25 -> 5; }

        { 1 -> 2; bandwidth= 1000000; latency= 5000; }
        { 2 -> 3; bandwidth= 100000; latency= 8000; }
        { 3 -> 4; bandwidth= 64000; latency= 800000; }
        { 4 -> 5; bandwidth= 1000000; latency= 10000; }
}
```

E.6 Scenario 6

```
nodes{
        default{
                function = SRC;
                start_time = 0,100;             # Start at 0 + 100 microseconds
                plot = true;                    # Plot this node
                num_pkts = 200;                 # Num packets to send
                }
```

```
        node 1 { function = ROUTER; plot=false; params= 'thresh 5 0.75'; }
        node 2 { function = ROUTER; params= 'thresh 5 0.75'; }
        node 3 { function = ROUTER; params= 'thresh 5 0.75'; }
        node 4 { function = ROUTER; params= 'thresh 5 0.75'; }
        node 5 { function = ROUTER; plot=false; params= 'thresh 5 0.75'; }

        node 8 { function = DST;  }
        node 10 { function = DST; }
        node 12 { function = DST; }
        node 14 { function = DST; }
        node 15 { function = DST; }

        node 6 { dest = 15; num_pkts = 1000; }
        node 7 { start_time =    1,0; dest =  8; }
        node 9 { start_time =   75,0; dest = 10; }
        node 11 { start_time = 110,0; dest = 12; }
        node 13 { start_time = 145,0; dest = 14; }
}

# Bandwidth in bps, latency in microseconds
edges{
        default{ bandwidth = 10000000; latency = 1; }
        { 1 -> 2; bandwidth = 64000; latency = 1000000;}
        { 2 -> 3; bandwidth = 64000; latency = 1000000;}
        { 3 -> 4; bandwidth = 64000; latency = 1000000;}
        { 4 -> 5; bandwidth = 64000; latency = 1000000;}
        { 6 -> 1; }
        { 7 -> 1; }
        { 8 -> 2; }
        { 9 -> 2; }
        { 10 -> 3; }
        { 11 -> 3; }
        { 12 -> 4; }
        { 13 -> 4; }
        { 14 -> 5; }
        { 15 -> 5; }
}
```

E.7 Scenario 7

```
nodes{
        default{
                function = SRC;                 # Source
                start_time = 0,100;             # Start at 0 + 100 microseconds
                plot = true;                    # Plot this node
                num_pkts = 1000;                # Num packets to send
                }

        node 1 { function = ROUTER; params= 'thresh 5 0.75'; }
        node 2 { function = ROUTER; params= 'thresh 5 0.75'; }
        node 3 { function = ROUTER; params= 'thresh 5 0.75'; }
        node 4 { function = ROUTER; params= 'thresh 5 0.75'; }
        node 9 { function = DST; }
        node 10 { function = DST; }
        node 11 { function = DST; }
```

```
        node 12 { function = DST; }

        node 5 { dest = 10; params= 'pktsize 70 1500'; }
        node 6 { start_time = 140,0; dest = 11;
                num_pkts= 100; params= 'pktsize 400 900'; }
        node 7 { start_time = 12,0; dest = 12; params= 'pktsize 700 1500'; }
        node 8 { start_time = 18,0; dest = 9; params= 'pktsize 900 1500';}
}

# Bandwidth in bps, latency in microseconds
edges{
        default{ bandwidth = 10000000; latency = 1; }
        { 1 -> 2; }
        { 2 -> 3; bandwidth = 64000; latency = 1000000;}
        { 3 -> 4; }
        { 5 -> 1; }
        { 6 -> 1; }
        { 7 -> 1; }
        { 8 -> 3; }
        { 9 -> 2; }
        { 10 -> 3; }
        { 11 -> 4; }
        { 12 -> 4; }
}
```

E.8 Scenario 8

```
nodes{
        default{
                function = SRC;                 # Source
                start_time = 0,100;             # Start at 0 + 100 microseconds
                plot = true;                    # Plot this node
                num_pkts = 1000;                # Num packets to send
                }

        node 1 { function = ROUTER;
                params='thresh 5 0.75 route 10 via 13 at 100.0'; }
        node 2 { function = ROUTER; params='thresh 5 0.75'; }
        node 3 { function = ROUTER;
                params='thresh 5 0.75  route 5 via 13 at 100.0'; }
        node 4 { function = ROUTER; plot=false; params='thresh 5 0.75'; }
        node 13 { function = ROUTER; params='thresh 5 0.75'; }
        node 9 { function = DST;  }
        node 10 { function = DST; }
        node 11 { function = DST; }
        node 12 { function = DST; }

        node 5 { dest = 10; }
        node 6 { start_time = 140,0; dest = 11; num_pkts= 100; }
        node 7 { start_time = 12,0; dest = 12;}
        node 8 { start_time = 18,0; dest = 9;}
}

# Bandwidth in bps, latency in microseconds
edges{
```

```
        default{ bandwidth = 10000000; latency = 1; }
        { 1 -> 2; }
        { 2 -> 3; bandwidth = 64000; latency = 1000000;}
        { 3 -> 4; }
        { 5 -> 1; }
        { 6 -> 1; }
        { 7 -> 1; }
        { 8 -> 3; }
        { 9 -> 2; }
        { 10 -> 3; }
        { 11 -> 4; }
        { 12 -> 4; }
        { 1 -> 13; bandwidth = 512000; latency = 10000;}
        { 3 -> 13; bandwidth = 512000; latency = 10000;}
}
```

Appendix F

Theoretical Rate Calculation for Scenario 2

Scenario 2 has a well-defined set of traffic flows in a network whose links have fixed bandwidths. It should be possible to determine a set of rates for each flow which ensures the highest link utilisation, where possible, and also divides the link capacity fairly amongst the flows.

In Scenario 2, the bottleneck in the system is the 64kbps full-duplex link 2 $\leftrightarrow$ 3. We will assume that all acknowledgment rates are limited by their respective data rates by

$$A_x = R * D_x,\ where\ R = \frac{ack\ pkt\ size}{data\ pkt\ size} = \frac{40}{1,500} \tag{F.1}$$

where D_x denotes a data flow from source x to its destination and A_x denotes the corresponding acknowledgment flow in the reverse direction.

At time $t = 0$, the flows D_5 and A_5 commence. The optimum rates for both are:

Time	**Optimum Rate (bps)**	
	D_5	A_5
$t \geq 0$	64,000	1706

At time $t = 12$, the flows D_7 and A_7 commence. As the link 2 $\leftrightarrow$ 3 is the bottleneck, we can write limiting equations for flows across the link 2 $\rightarrow$ 3 and across the link 3 $\rightarrow$ 2:

$$D_5 + D_7 \leq 64,000 \tag{F.2}$$
$$A_5 + A_7 \leq 64,000 \tag{F.3}$$

(2) is the limiting equation, as the data flows are substantially larger than the acknowledgment flows. By sharing bandwidth equally to the flows, we obtain the rates:

Time	**Optimum Rate (bps)**			
	D_5	A_5	D_7	A_7
$t \geq 12$	32,000	853	32,000	853

At time $t = 18$, the flows D_8 and A_8 commence. Here, the flows are in the opposite directions to the previous four flows. Again, we can write limiting equations for the flows:

$$D_5 + D_7 + A_8 \leq 64,000 \tag{F.4}$$
$$A_5 + A_7 + D_8 \leq 64,000 \tag{F.5}$$

Distributing bandwidth fairly, we have $D_5 = D_7$ and $A_5 = A_7$. Assume both links are fully utilised and substituting for the acknowledgment rates using (1), we have:

$$2D_5 + R * D_8 = 64,000 = 2R * D_5 + D_8 \tag{F.6}$$
$$2D_5(1 - R) = D_8(1 - R) \tag{F.7}$$
$$2D_5 = D_8 \tag{F.8}$$

Substituting (8) into (6) gives $D_5 = 31,168$, $D_8 = 62,337$ and the table:

Time	Optimum Rate (bps)					
	D_5	A_5	D_7	A_7	D_8	A_8
$t \geq 18$	31,168	831	31,168	831	62,337	1,662

The six flows continue to transmit until time $t = 140$ when the flows D_6 and A_6 commence. Again, distributing bandwidth fairly, we have $D_5 = D_7 = D_6$ and $A_5 = A_7 = A_6$. Assume both links are fully utilised and substituting for the acknowledgment rates using (1), we have:

$$3D_5 + R * D_8 = 64,000 = 3R * D_5 + D_8 \tag{F.9}$$
$$3D_5(1 - R) = D_8(1 - R) \tag{F.10}$$
$$3D_5 = D_8 \tag{F.11}$$

Substituting (11) into (9) gives $D_5 = 20,779$, $D_8 = 62,337$ and the table:

Time	Optimum Rate (bps)							
	D_5	A_5	D_7	A_7	D_8	A_8	D_6	A_6
$t \geq 140$	20,779	554	20,779	554	62,337	1,662	20,779	554

As flow D_6 only has 100 packets to send, it finishes quickly, and the rates return to the values before flow D_6 started:

Time	Optimum Rate (bps)					
	D_5	A_5	D_7	A_7	D_8	A_8
$t \geq Off_6$	31,168	831	31,168	831	62,337	1,662

Similarly, flow D_8 has most of the bandwidth across the link $3 \rightarrow 2$, so it is the next to finish, and the remaining flows return to the rate values before flow D_8 started:

Time	Optimum Rate (bps)			
	D_5	A_5	D_7	A_7
$t \geq Off_8$	32,000	853	32,000	853

The flows D_5, D_7, A_5 and A_7 continue at these rates until one or both finish.

Appendix G

Obtaining the REAL Simulator

The modified version of the REAL 4.0 simulator which was used to obtain the results in this thesis, and the scenarios simulated in this thesis, can be obtained as follows:

Via the World Wide Web: At http://minnie.cs.adfa.oz.au/PhD/

Via Anonymous File Transfer: Ftp to `minnie.cs.adfa.oz.au` [131.236.21.160]. Login either as `anonymous` or `ftp`; use your email address as your password. Change to the `PhD` folder.

The following files should be available for you to retrieve:

wkt_real.tar.gz A full distribution of REAL 4.0, with modifications as described in Chapter 8. Note that the modifications have only been verified on machines running SunOS 3.5 and FreeBSD 2.x.

wkt_scenarios.tar.gz The input scenario files which were discussed in Chapter 9, and listed in Appendix E. Please note the rider at the beginning of Appendix E.

random_scenarios.tar.gz The 500 randomly-generated scenario files which were described in Chapter 10.

param_changes.tar.gz A set of changes which were made to the 500 randomly-generated scenario files in order to obtain the parameter results described in Chapter 11.

trump_spin.tar.gz The input files to the Spin protocol verification suite which were used to verify the TRUMP protocol, described in Appendix B.

All files are compressed using GNU zip, and you will need GNU `gunzip` to decompress them. All files are Tar file archives, and you will need a `tar` program to extract the files from the archives. If you have any problems with these files, please email me at `wkt@cs.adfa.oz.au`.

Any changes or notes about these files which did not make it in to this thesis will be placed in a `README` file. Note that the modifications to REAL 4.0, the input scenario files and the thesis file are Copyright

Bibliography

[Ahn *et al* 95] J. S. Ahn, P. Danzig, Z. Liu, and L. Yan. "Evaluation of TCP Vegas: Emulation and Experiment". In *Proceedings of the 1995 SIGCOMM Conference*, pp 185–195. August 1995.

[Ait-Hellal & Altman 96] O. Ait-Hellal and E. Altman. "Problems in TCP Vegas and TCP Reno". In *Congress De Nouvelles Architectures pour les Communications*. Held in Paris, France. December 1996.

[Apisdorf *et al* 96] J. Apisdorf, K. Claffy, Kevin Thompson, and Rick Wilder. "OC3MON: Flexible, Affordable, High Performance Statistics Collection". In *USENIX 10th Systems Administration Conference*, pp 97–112. September 1996.

[Bates 98] T. Bates. "The Cidr Report". URL http://www.employees.org/~tbates/cidr.plot.html. 1998.

[Benmohamed & Meerkov 93] L. Benmohamed and S. M. Meerkov. "Feedback Control of Congestion in Packet Switched Networks: The Case of a Single Congested Node". *IEEE Transactions on Networking*, Vol. 1, No. 6, pp 693–707. December 1993.

[Bennett & Des Jardins 94] J. Bennett and G. T. Des Jardins. "Comments on the July PRCA Rate Control Baseline". Technical Report 94-0682, ATM Forum Traffic Management subworking group. July 1994.

[Braden & Postel 87] R. Braden and J. Postel. "Requirements for Internet Gateways". RFC 1009. January 1987.

[Brakmo & Peterson 95] L. Brakmo and L. Peterson. "Performance Problems in BSD4.4 TCP". *Computer Communication Review*, Vol. 25, No. 5, pp 69–86. October 1995.

[Brakmo *et al* 94] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. "TCP Vegas: New Techniques for Congestion Detection and Avoidance". In *Proceedings of the 1994 SIGCOMM Conference*, pp 24–35. August 1994.

[Cerf & Kahn 74] V. Cerf and R. Kahn. "A Protocol for Packet Network Intercommunications". *IEEE Transactions on Communications*, Vol. 22, No. 5, pp 637–648. May 1974.

[Charney 94] A. Charney. "An Algorithm for Rate Allocation in a Packet-Switched Network with Feedback". M.Sc. thesis, Department of EECS, MIT. May 1994.

[Charney *et al* 94] A. Charney, D. Clark, and R. Jain. "Congestion Control with Explicit Rate Indication". Technical Report 94-0692, ATM Forum Traffic Management subworking group. July 1994.

[Cheriton & Williamson 89] D. Cheriton and C. Williamson. "VMTP as the Transport Layer for High-Performance Distributed Systems". *IEEE Communications Magazine*, pp 37–44. June 1989.

[Cheriton 86] D. Cheriton. "VMTP: A Transport Protocol for the Next Generation of Communication Systems". In *Proceedings of the 1986 SIGCOMM Conference*, pp 406–415. August 1986.

[Clark 82] D. D. Clark. "Window and Acknowledgment Strategy in TCP". RFC 813. July 1982.

[Clark 88] D. D. Clark. "The Design Philosophy of the DARPA Internet Protocols". In *Proceedings of the 1988 SIGCOMM Conference*, pp 106–114. August 1988.

[Clark *et al* 87] D. Clark, M. Lambert, and L. Zhang. "NETBLT: A High Throughput Transport Protocol". In *Proceedings of the 1987 SIGCOMM Conference*, pp 353–359. August 1987.

[Comer 88] D. Comer. "Internetworking with TCP/IP — Principles, Protocols and Architecture, 1st Edition". Prentice-Hall, ISBN 0-13-470188-7. 1988.

[Comer 95] D. Comer. "Internetworking with TCP/IP — Principles, Protocols and Architecture, 3rd Edition". Prentice-Hall, ISBN 0-13-216987-8. 1995.

[Davin & Heybey 90] J. R. Davin and A. T. Heybey. "A Simulation Study of Fair Queueing and Policy Enforcement". *Computer Communication Review*, Vol. 20, No. 5, pp 23–29. October 1990.

[Davis 72] D. W. Davis. "The Control of Congestion in Packet-Switching Networks". *IEEE Transactions on Communications*, Vol. 20, No. 6, pp 547–550. June 1972.

[Degermark *et al* 97] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. "Small Forwarding Tables for Fast Routing Lookups". In *Proceedings of the 1997 SIGCOMM Conference*, pp 3–14. August 1997.

[Demers *et al* 89] A. Demers, S. Keshav, and S. Shenker. "Analysis and Simulation of a Fair Queueing Algorithm". In *Proceedings of the 1989 SIGCOMM Conference*, pp 1–12. September 1989.

[Eldridge 92] C. A Eldridge. "Rate Controls in Standard Transport Layer Protocols". *Computer Communication Review*, Vol. 22, No. 3, pp 106–120. July 1992.

[Fendick *et al* 92] K. W. Fendick, M. A. Rodruigues, and A. Weiss. "Analysis of a Rate-Based Control Strategy with Delayed Feedback". In *Proceedings of the 1992 SIGCOMM Conference*, pp 136–147. September 1992.

[Fletcher 82] J. Fletcher. "An Arithmetic Checksum for Serial Transmissions". *IEEE Transactions on Communications*, Vol. 30, No. 1, pp 247–252. January 1982.

[Floyd & Jacobson 91] S. Floyd and V. Jacobson. "Traffic Phase Effects in Packet-Switched Networks". *Computer Communication Review*, Vol. 21, No. 2, pp 26–42. April 1991.

[Floyd & Jacobson 93] S. Floyd and V. Jacobson. "Random Early Detection Gateways for Congestion Avoidance". *ACM Transactions on Networks*, Vol. 1, No. 4, pp 397–413. August 1993.

[Haas 91] Z. Haas. "Adaptive Admission Congestion Control". *Computer Communication Review*, Vol. 21, No. 5, pp 58–76. October 1991.

[Hashem 90] E. Hashem. "Random Drop Congestion Control". M.Sc. thesis, Department of CS, MIT. 1990.

[Hluchyj 94] M. Hluchyj. "Closed-Loop Rate-Based Traffic Management". Technical Report 94-0211R3, ATM Forum Traffic Management subworking group. April 1994.

[Holzmann 90] G. J. Holzmann. "Algorithms for Automated Protocol Verification". *AT&T Technical Journal*, Vol. 69, No. 1, pp 32–44. January 1990.

[Holzmann 91] G. Holzmann. "Design and Validation of Computer Protocols". Prentice-Hall, ISBN 0-13-539834-7. 1991.

[Holzmann 97] G. Holzmann. "The Model Checker Spin". *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp 279–295. May 1997.

[Jacobson 88] V. Jacobson. "Congestion Avoidance and Control". In *Proceedings of the 1988 SIGCOMM Conference*, pp 314–329. August 1988.

[Jacobson 90] V. Jacobson. "Compressing TCP/IP headers for low-speed serial links". RFC 1144. February 1990.

[Jain & Ramakrishnan 88] R. Jain and K. K. Ramakrishnan. "Congestion Avoidance in Computer Networks with a Connectionless Network Layer: Concepts, Goals and Methodology". In *Proceedings of the 1988 Computer Networking Symposium*, pp 134–143. April 1988.

[Jain 86] R. Jain. "A Timeout-Based Congestion Control Scheme for Window Flow-Controlled Networks". In *Innovations in Internetworking*, pp 289–293. Artech House. 1986.

[Jain 90] R. Jain. "Congestion Control in Computer Networks: Issues and Trends". *IEEE Network Magazine*, Vol. 4, No. 3, pp 24–30. May 1990.

[Jain 96] R. Jain. "Congestion Control and Traffic Management in ATM Networks: Recent Advances and a Survey". *Computer Networks and ISDN Systems*, Vol. 28, No. 13, pp 1723–1738. October 1996.

[Jain *et al* 87] R. Jain, K. K. Ramkrishnan, and D. M. Chiu. "Congestion Avoidance in Computer Networks with a Connectionless Network Layer". Technical Report TR-506, Digital Equipment Corporation. 1987.

[Kalmanek *et al* 90] Kalmanek, Kanakia, and Keshav. "Rate Controlled Servers for Very High Speed Networks". In *Proceedings of GlobeCom '90*. 1990.

[Karn & Partridge 87] P. Karn and C. Partridge. "Improving Round-Trip Time Estimates in Reliable Transport Protocols". In *Proceedings of the 1987 SIGCOMM Conference*, pp 2–7. August 1987.

[Keshav 88] S. Keshav. "REAL: A Network Simulator". Technical Report 88/472, Department of EECS, UC Berkeley. 1988.

[Keshav 91] S. Keshav. "Congestion Control in Computer Networks". PhD thesis, Department of EECS, UC Berkeley. August 1991.

[Majithia & *et al* 79] J. C. Majithia *et al*. "Experiments in Congestion Control Techniques". In *Proceedings of the International Symposium on Flow Control in Computer Networks*, pp 211–234. February 1979.

[Mankin & Ramakrishnan 91] A. Mankin and K. Ramakrishnan. "Gateway Congestion Control Survey". RFC 1254. August 1991.

[Mankin 90] A. Mankin. "Random Drop Congestion Processing". In *Proceedings of the 1990 SIGCOMM Conference*, pp 1–29. September 1990.

[Mishra *et al* 96] P. P. Mishra, H. Kanakia, and S. K. Tripathi. "On Hop-by-Hop Rate-Based Congestion Control". *IEEE/ACM Transactions on Networking*, Vol. 4, No. 2, pp 224–239. April 1996.

[Morrison 68] D. Morrison. "PATRICIA – Practical Algorithm to Retrieve Information Coded In Alfanumeric". *Journal of the ACM*, Vol. 15, No. 4, pp 514–534. October 1968.

[Nagle 84] J. Nagle. "Congestion Control in IP/TCP Internetworks". RFC 896. January 1984.

[Nagle 87] J. Nagle. "On Packet Switches with Infinite Storage". *IEEE Transactions on Communications*, Vol. 35, No. 4, pp 435–438. April 1987.

[Newman 94] P. Newman. "Traffic Management for ATM Local Area Networks". *IEEE Communications Magazine*, Vol. 32, No. 8, pp 44–50. August 1994.

[Partridge 94] C. Partridge. "Gigabit Networking", chapter 4, pp 61–87. Addison-Wesley. ISBN 0-201-56333-9. 1994.

[Postel 80] J. B. Postel. "User Datagram Protocol". RFC 768. August 1980.

[Postel 81a] J. B. Postel. "Internet Control Message Protocol". RFC 792. September 1981.

[Postel 81b] J. B. Postel. "Transmission Control Protocol". RFC 793. September 1981.

[Ramakrishnan & Jain 90] K. K. Ramakrishnan and R. Jain. "A Binary Feedback Scheme for Congestion Avoidance in Computer Networks". *ACM Transactions on Computer Systems*, Vol. 8, No. 2, pp 158–181. May 1990.

[Reynolds & Postel 87] J. Reynolds and J. Postel. "The Request For Comments Reference Guide". RFC 1000. August 1987.

[Roberts 94] L. Roberts. "Enhanced PRCA (Proportional Rate-Control Algorithm)". Technical Report 94-0735R1, ATM Forum Traffic Management subworking group. August 1994.

[Robinson *et al* 90] J. Robinson, D. Friedman, and M. Steenstrup. "Congestion Control in BBN Packet-Switched Networks". *Computer Communication Review*, Vol. 20, No. 1, pp 76–90. January 1990.

[Rose 92] O. Rose. "The Q-Bit Scheme". *Computer Communication Review*, Vol. 22, No. 2, pp 29–42. April 1992.

[Sanders & Weaver 90] R. M. Sanders and A. C. Weaver. "The Xpress Transfer Protocol (XTP) — A Tutorial". *Computer Communication Review*, pp 67–80. October 1990.

[Schwartz 82] M. Schwartz. "Performance Analysis of the SNA Route Pacing Control". *IEEE Transactions on Communications*, Vol. 30, No. 1, pp 172–184. January 1982.

[Shenker 94] S. Shenker. "Making Greed Work in Networks: A Game-Theoretic Analysis of Switch Service Disciplines". In *Proceedings of the 1994 SIGCOMM Conference*, pp 47–57. September 1994.

[Sidi *et al* 93] M. Sidi, W. Liu, I. Cidon, and I. Gopal. "Congestion Control Through Input Rate Regulation". *IEEE Transactions on Communications*, Vol. 41, No. 3, pp 471–477. March 1993.

[Stallings 91] W. Stallings. "Data and Computer Communications, 3rd Edition". Maxwell Macmillan International, ISBN 0-02-946478-1. 1991.

[Stallings 96] W. Stallings. "IPv6: The New Internet Protocol". *IEEE Communications Magazine*, Vol. 34, No. 7, pp 96–108. July 1996.

[Stanford 96] U. Stanford. "Stanford University Workshop on Fast Routing and Switching". URL http://tiny-tera.stanford.edu/Workshop_Dec96/. 1996.

[Tichy 87] W. F. Tichy. "An Introduction to the Revision Control System". In *Unix Programmer's Manual: Supplementary Documents 1*, chapter 13. USENIX Association. June 1987.

[Turner 86] J. Turner. "New Directions in Communications". *IEEE Communications Magazine*, Vol. 24, No. 10, pp 8–15. October 1986.

[Vahalia 96] U. Vahalia. "UNIX Internals: The New Frontiers", chapter 5, pp 114–115. Prentice-Hall. ISBN 0-13-101908-2. 1996.

[Wang & Crowcroft 81] Z. Wang and J. Crowcroft. "Eliminating Periodic Packet Loss in the 4.3-Tahoe BSD TCP Congestion Control Algorithm". *Computer Communication Review*, Vol. 11, No. 1, pp 9–16. January 1981.

[Wang & Crowcroft 91] Z. Wang and J. Crowcroft. "A New Congestion Control Scheme: Slow Start and Search (Tri-S)". *Computer Communication Review*, Vol. 21, No. 1, pp 32–43. January 1991.

[Williamson & Cheriton 91] C. L. Williamson and D. R. Cheriton. "Loss-Load Curves: Support for Rate-Based Congestion Control in High-Speed Datagram Networks". In *Proceedings of the 1991 SIGCOMM Conference*, pp 17–28. September 1991.

[Zhang 86] L. Zhang. "Why TCP Timers Don't Work Well". In *Proceedings of the 1986 SIGCOMM Conference*, pp 397–405. August 1986.

[Zhang 91] L. Zhang. "VirtualClock: A New Traffic Control Algorithm For Packet-Switched Networks". *ACM Transactions on Computer Systems*, Vol. 9, No. 2, pp 101–124. May 1991.